AN ABSTRACT OF THE DISSERTATION OF

Peter Byerley Rindal for the degree of Doctor of Philosophy in Computer Science presented on August 16, 2018.

Title: Keeping your Friends Secret: Improving the Security, Efficiency and Usability of Private Set Intersection.

Abstract approved: ______________________________

Mike Rosulek

Private set intersection (PSI) allows two parties, who each hold a set of items, to compute the intersection of those sets without revealing anything about other items. Recent advances in PSI have significantly improved its performance for the case of semi-honest security, making semi-honest PSI a practical alternative to insecure methods for computing intersections. However, these protocols have two major drawbacks: 1) the amount of data required to be communicated can be orders of magnitude larger than an insecure solution and 2) when in the presence of malicious parties the security of these protocols breaks down.

In this work, four malicious secure PSI protocols are introduced along with three semi-honest protocols which have sublinear communication. These protocols are based on a combination of fast symmetric-key primitives and fully homomorphic encryption. Three of these protocols represent the current state of the art for their respective settings.

The practicality of these protocols are demonstrated with prototype implementations. To securely compute the intersection of two sets of size 2^{20} in the malicious setting requires only 13 seconds, which is $\sim 450\times$ faster than the previous best malicious-secure protocol (De Cristofaro et al, Asiacrypt 2010), and only $3\times$ slower than the best semi-honest protocol (Kolesnikov et al., CCS 2016). Alternatively, when computing the intersection between set sizes of 2^{10} and 2^{28}, our fastest protocol require just 6 seconds and 5MB of communication.

Keeping your Friends Secret: Improving the Security, Efficiency and Usability of Private Set Intersection

by
Peter Byerley Rindal

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented August 16, 2018
Commencement June 2019

Doctor of philosophy dissertation of Peter Byerley Rindal presented an August 16, 2018.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

Head of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Peter Byerley Rindal, Author

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation to:

- Mike Rosulek for his superb guidance and patience. It has been inspiring to observe and attempt to emulate the excellence that he brings to the field of Cryptography.

- My coauthors which include Hao Chen, Daniel Demmler, Kim Laine, Kristin Lauter, Ni Trieu and Huang Zhicong. This dissertation would not have been possible without their expertise.

- My family and friends for without their support, this journey would have been far longer. In particular, I would like to thank my father for his endless interest in my endeavors and encouraging me to reach further.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

Page

Chapter 1

Introduction

1.1 Problem Statement

The problem of identifying the items in common between two or more sets often arises in a variety of real-world situations. For example, when people meet it is often useful to identify mutual friends. Digital approaches to solving this problem have traditionally required both sets to be represented on a single computer where the items in common can be identified. While this solution works in many situations, privacy concerns are increasingly making such data aggregation difficult. Consider the case where two parties each possess a set which contain sensitive information. In this situation, it is often undesirable to bring the sets together on a single computer as this could allow one of the parties to learn too much information.

To address these privacy concerns a series of techniques broadly known as Private Set Intersection (PSI) have been developed. These techniques assume that it is desirable to learn the items in common (the intersection) while hiding all other items. For simplicity let us consider the two-party case with Alice and Bob where each holds a respective set X, Y. The requirements of PSI state that Bob, and possibly Alice too, should learn the intersection of these two sets $X \cap Y$ and nothing more.

Conceptually, PSI can be thought of as emulating a trusted third party that receives both of the sets X, Y, privately informs Bob of the intersection $X \cap Y$ and then forgets everything. The central difference between a trusted third party and PSI is that Alice & Bob directly interact with each other without any assistance. At the end of the protocol Bob learns the intersection and Alice learns nothing.

Initially, it may seem impossible for Bob to identify the items in common between the sets while learning nothing more. However, with the development of modern cryptographic techniques, this functionality can be achieved with strong mathematical proofs of correctness and privacy. More generally, it has been shown that any function $f(x_1, ..., x_n)$ can be jointly computed between n parties, each with a private input x_i, such that the parties learn the result of the function and nothing more. This general setting is known as secure multi-party computation (MPC) and PSI is a special case where the intersection function is computed. While MPC implies the feasibility of PSI, the concrete performance of these general-purpose techniques prevents many practical applications. This thesis presents several purpose-built PSI protocols that bridge this gap in performance.

One of two security notions is often considered by MPC protocols. The simplest is semi-honest security where the participating parties honestly follow the protocol instruction but try to infer additional information beyond their specified output. This setting is often sufficient when the other party somewhat trusted and their cost of getting caught cheating is high. A much stronger notion is known as malicious security. In this setting, the corrupt party is assumed to arbitrarily deviate from the protocol specification, possibly with the goal of gaining additional information or influencing the output of the other party. See Section 2.1 for additional details on these security notions.

1.2 Applications

To further motivate PSI we now detail several of the most promising applications and why a solution without privacy is unacceptable. Several of these applications bring specific requirements above and beyond the generic problem of PSI which inform current and future research directions.

1.2.1 Voter Registration

United States citizens can register to vote in any state which they claim residency. In the event that someone moves between states, it is possible for them to become registered to vote in both states. This individual should then unregister themselves from the old state. However, for a variety of reasons, this does not always happen. If for some reason this individual casts a vote in both states, either on purpose or by accident, the outcome of the vote may be incorrect.

To catch cases of such double voting, states would need to compare the lists of registered voters. However, these lists contain highly sensitive information and states are reluctant to share them. In particular, this information could enable a wide range of undesirable political maneuvers aimed at providing an unfair advantage to one side of an election.

An existing solution to this problem is the Electronic Registration Information Center (ERIC)[Ham]. This organization acts as a trusted third party that is used to identify incorrectly registered individuals. As a result, this single organization holds a large amount of highly sensitive information and represents a single point of failure should a data breach occur. An alternative and more secure approach would be to remove this organization and perform PSI directly between the states to identify incorrectly registered citizens. Once these individuals are identified, an offline procedure could be initiated to unregister them from one of the states. This general approach is also used to update information such as the current address of a voter.

1.2.2 Contact Discovery

Another highly motivated application is known as contact discovery. In this scenario, a new user has just registered on a social networking service. For example, let us consider the messaging app known as WhatsApp which offers a service similar to texting except that the messages are encrypted between the sender and recipient. To increase the usability of this secure messaging service, the user's contacts are uploaded to the WhatsApp servers. Here the intersection between the user's contacts and all WhatsApp users is computed and automatically added to the user's app as WhatsApp contacts.

While this conveniently alleviates the user from having to manually add their contacts, the service provider now learns significant and potentially very private information about the social graph of their users. Perhaps most troubling is that users who never sign up for the service still have significant information about them held by the service provider, e.g. their name, phone number, email, friends, and possibly much more. Recently the German courts agreed that this level of privacy loss is unacceptable and that WhatsApp must obtain written permission from each of the contacts before their information can be updated to their server[Pos17].

A perfect solution is to use PSI between the user and the service provider. Here the user would be able to identify all of their contacts that also use the service while preventing the service provider from learning information about contacts that have not signed up for the service. To address this specific use case Section 1.3.4 details a current research direction which promises to give practical performance.

1.2.3 Threat Log Comparison

Our current computer networks are constantly under attack by malicious parties that try to learn sensitive information or adversely affect the system. To prevent such behavior network administrators utilize a variety of safeguards such as firewalls. Another technique is to log all the activity on the network and attempt to identify anomalies and common attacks. However, if the attack is distributed across several networks any one of them may not be able to identify it or realize how pervasive of a threat it is. By aggregating the data together, these attacks can become easier to identify and stop. However, simply sharing logs in the clear can itself be a security concern due to a large amount of information contained within them.

As with the voter registration example, existing solutions utilize trusted third parties. PSI or a closely related variant could replace the trusted third party. In this case, two or more parties can come together to identify common threats by computing the intersection of the attacks that all, or some, parties have observed.

1.2.4 Ad-Revenue Conversion

One of the few applications of PSI currently being used is to compute the ad-revenue conversion rate. This application is deployed by Google to compute how effective their online advertising is at generating offline sales[IKN+17a]. For example, say Tesla has purchased ads on Google search. Google will, therefore, hold a list of users who have viewed the ad while Tesla has a list of people who purchased their cars. Tesla would like to learn how many users both saw the ad and purchased a car. Knowing this provides an important metric for deciding how effective their advertising was.

Importantly, neither of these companies are willing to reveal their lists. Instead Google and Tesla engage in a PSI protocol that compute the *size* of the intersection along with the total amount of money these users spent. While not exactly PSI, the general approach their system takes is a natural extension of many existing PSI solutions.

1.3 Contributions

This dissertation improves the state-of-the-art private set intersection techniques as follows.

1. Strengthens semi-honest secure private set intersection protocols to be secure in the malicious model.
2. Improves protocol performance through the application of more efficient primitives.
3. Richer functionality that more accurately captures the requirements of real-world applications.

Given the wide range of motivating application for PSI, achieving these contributions represents a significant advance. Most notable is the development of the two most efficient malicious secure PSI protocols which are based on the Bloom filter and hash table data structures. Concurrently, the first two practical PSI protocol, based on fully homomorphic encryption and private information retrieval have also been proposed and are distinguished due to them achieving sub-linear communication.

A overview of the contained protocols is as follows.

1.3.1 Malicious Secure Bloom Filter PSI

Dong, Chen & Wen [DCW13] introduced a PSI technique uses a data structure known as Bloom filter[ABPH07] and derives its security from the highly efficient oblivious transfer extension primitive[IKNP03, KOS15a]. This protocol was presented in the semi-honest setting along with an extension that was claimed to be malicious secure. This turns out not to be true with several attacks being identified in Chapter 3 and concurrently by [Lam16]. In addition to identifying security flaws, Chapter 3 contains a malicious secure solution based on a protocol technique known as cut-and-choose.

The PSI protocol works by first having the two parties construct a Bloom filter for their respective sets. A Bloom filter is a long bit vector which is initially set to all zeros. To insert an items x the bits indexed by random functions $h_1(x), ..., h_k(x)$ are set to one. The Bloom filter can then be used to test if some y is contained in the data structure by checking if all of the corresponding bits are also set to one. For sufficiently large parameters, this procedure will produce the correct result with overwhelming probability.

Dong et al. [DCW13] adapted this procedure to ensure that the only information revealed is a single bit denoting the result of the set membership test, $y \overset{?}{\in} X$. This is achieved by replacing the bit vector with a vector of random κ-bit values $\{m_1, ..., m_n\}$ chosen by Alice. Bob who receives output and holds set Y is allowed to learn all messages m_j such that $j \in \{h_i(y) \mid y \in Y, i \in [k]\}$. A randomized encoding is then defined as,

$$[\![y]\!] := \bigoplus_{i \in [k]} m_{h_i(y)}$$

Importantly, via another procedure utilizing oblivious transfer, Bob *only* learns these messages needed to compute $[\![y]\!]$ for all $y \in Y$ while Alice is oblivious to which these are. For a sufficiently large Bloom filter, it can be ensured that for all $y' \notin Y$ there is an overwhelming probability that at least one of the messages m_j needed to compute $[\![y']\!]$ is unknown to Bob. Therefore $[\![y']\!]$ is unknown to Bob and uniformly distributed from his view.

The PSI protocol is completed by having Alice compute all $[\![x]\!]$ for $x \in X$ and send these values to Bob who infers the intersection of $X \cap Y$ from the intersection of $\{[\![x]\!] \mid x \in X\} \cap \{[\![y]\!] \mid y \in Y\}$. As previously implied, for all $x \in (X \backslash Y)$ the encoding $[\![x]\!]$ is uniformly distributed in Bob's view and therefore any encoding of an x not in the intersection reveals no information about the value of $x \in (X \backslash Y)$, i.e. $\mathsf{Encode}(x) \rightarrow [\![x]\!]$ is a one-way function.

In the semi-honest protocol presented by Dong et al.[DCW13] there is a trivial attack where a malicious Bob can learn all the messages m_j instead of just the messages needed to compute $\{[\![y]\!] \mid y \in Y\}$. In this case the argument that $\mathsf{Encode}(x)$ is one-way for Bob no longer holds. Dong et al. proposed a countermeasure to this attack but unfortunately, it too was not secure and very expensive to implement. After identifying the security bug, Chapter 3 details an improved protocol and implementation using a highly efficient technique to stop such attacks. The primary idea of this technique is to make Bob approximately prove that many of the messages are unknown to him. In particular, we check certain properties on roughly 1% of the messages and from this infer that many messages in the remaining 99% are unknown to Bob. Because so few messages are actually checked the overhead of this technique is marginal compared to the semi-honest version of the protocol. Chapter 3 details this protocol and demonstrates that it achieves very good performance compared to the weaker semi-honest variant of Dong et al.[DCW13]. The protocol was published at Eurocrypt 2017 and written by Rindal & Rosulek under the title "Improved Private Set Intersection against Malicious Adversaries"[RR17a].

1.3.2 Malicious Secure Dual Execution PSI

For set sizes larger than $n = 100,000$ the Bloom filter approach above starts to have impractical performance due to a large amount of communication. Even in the semi-honest setting, the Bloom filter technique requires roughly $2\kappa^2 n = 32,768n$ bits of communication to compute the intersection. However, a different paradigm proposed by Pinkas, Schneider & Zohnar [PSZ14] achieves semi-honest security and only requires $5\kappa n = 640n$ bits of communications. Chapter 4 strengthens the protocol of Pinkas et al. [PSZ14, PSSZ15, PSZ18] to the malicious setting with a small added overhead.

This approach utilizes a hash table in lieu of a Bloom filter and employs a different construction to produce a randomized encoding of x. In particular, under a given encoding Bob, who learns the intersection, is allowed to compute $[\![y]\!]$ for a single value y while Alice is free to compute $[\![x]\!]$ for any x. This restriction differs from the Bloom filter approach where Bob could learn many encodings. Given this, a protocol to test if a single $y \in X$ is immediate. Bob computes $[\![y]\!]$ and Alice sends $\{[\![x]\!] \mid x \in X\}$ to Bob who infers the membership test $y \in X$ from the test $[\![y]\!] \in \{[\![x]\!] \mid x \in X\}$. As before Bob learning the randomizing encoding $[\![y]\!]$ does not reveal any information besides the desired output due to all encoding besides $[\![y]\!]$ being uniformly distributed in Bob's view.

The central difference between this approach and Bloom filter based protocol is that Bob can only learn a *single* randomized encoding instead of many. Given this limitation, one method of performing PSI between sets X and Y is to repeat the test for each $y \in Y$. However, this would lead to an overall complexity of $O(n^2)$ which is impractical for $n > 10,000$. Pinkas et al. [PSZ14] show how this approach can be augmented with a hash table data structure to reduce the complexity to be linear in n. This is achieved by constructing a vector of n bins where every $x \in X$ (and $y \in Y$ respectively) is placed in the bin indexed by a random function $h(x)$. In expectation, each bin will have one item and with overwhelming probability, no bin will have more than $O(\log n)$ items. To compute the intersection between X and Y, it is then sufficient to compute the intersection between each pair of bins. That is, for each $x = y$ it holds that both will be placed in the bin indexed by $h(x) = h(y)$ and therefore be contained in the intersection of that bin. By performing a quadratic cost intersection within each bin the overall complexity is $O(n \log^2 n)$[1]. This can further be reduced by using $n/\log n$ bins which results in a complexity of $O(n \log n)$ due to the maximum bin size increasing only by a small constant. Pinkas et al. [PSZ14] also show a more efficient hashing technique that further reduces the complexity to $O(n)$.

While this approach is highly efficient, it fails to achieve malicious security. Several issues arise in proving the security of this protocol. The largest issue is that for each bin, $O(\log n)$ membership tests are performed where Alice is free to use different inputs to each. This turns out to break the simulation-based proof of security due to it not corresponding to any honest input. The protocol of Chapter 4 overcomes this attack using a technique known as dual execution. At a very high-level this approach works by running the protocol twice in opposite directions and combining the final step of each execution. First, observe that Bob can only learn $\mu = O(\log n)$ encodings $\{[\![x]\!]_{\mathsf{B}}\}$, one for each of the μ membership tests. By running the encoding/membership protocol in both directions, Alice too can learn at most μ encodings $[\![y]\!]_{\mathsf{A}}$. We then define a common encoding $[\![z]\!] := [\![z]\!]_{\mathsf{B}} + [\![z]\!]_{\mathsf{A}}$. Alice is now restricted to knowing μ common encodings for this bin as desired. The protocol can securely be completed by Alice sending the common encodings $\{[\![y]\!] \mid y \in Y\}$ to Bob who the infers the intersection of X and Y.

With additional optimizations we demonstrate that our "dual execution" protocol is only 3× slower than the semi-honest variant when comparing sets of size $n = 1,000,000$, requiring a total of 12 seconds on a server. To date,

[1] For security reasons all bins must be padded with dummy items to their maximum size of $O(\log n)$ items. Hence comparing a single pair of bins has $O(\log^2 n)$ complexity.

this represents the fastest malicious secure protocol and is 450 times faster than the protocol of [DCKT10]. This protocol was authored by Rindal & Rosulek under the title "Malicious-Secure Private Set Intersection via Dual Execution" and was published at the 24rd ACM Conference on Computer and Communications Security (CCS 2017).

1.3.3 Malicious Secure PSI with Differential Privacy

Chapter 5 improves on the efficiency of the dual execution PSI protocol by allowing a bounded amount of information to be leaked. The mechanism uses a technique known as differential privacy which allows a tightly characterized and bounded amount of information to be leaked. In the dual execution PSI protocol the items in each set are first mapped to $m = O(n/\log n)$ bins using a publicly known hash function $h : \{0,1\}^* \to \{1, 2, ..., m\}$. A smaller PSI protocol is then applied to the matching bins. Observe that learning the number of items l_i in the ith bin reveals that the other party's set S has the property that $|\{s \in S \mid h(s) = i\}| = l_i$. In most settings, this amount of leakage is considered unacceptable. The original dual execution protocol avoids leaking this information by "padding" all bins with dummy items to a publicly known upper bound on l_i. In practice, this results in each bin containing three dummy items per real item.

Instead of padding to the maximum possible bin size, the protocol of Chapter 5 reveals a randomized estimate between the true bin size l_i and its upper bound. Revealing this *noisy* estimate allows a smooth trade-off between improved performance and information leakage. This information allows the protocol to utilizes up to three times fewer dummy items while still allowing the analysis to tightly bound the amount of information that is revealed. In practical terms, these improvements offer up to a 2× running time speedup. This protocol was authored by Groce, Rindal & Rosulek under the title "Cheaper Private Set Intersection via Differentially Private Leakage."

1.3.4 Fully Homomorphic Encryption based PSI

One setting where the protocols above fail to achieve good performance compared to an insecure solution is when Bob's set is much smaller than Alice's. In this case, the communication complexity remains proportional to the larger set. However, an insecure solution where the sets can be exchanged in the clear has communication complexity that scales with the smaller set. Consider the contact discovery application where Bob just signed up for a social networking service and wishes to have his existing contacts automatically populated in this app. To do this with the techniques above, the communication will be proportional to the larger set which could be upwards of a hundred million entries. This would result in the cellphone downloading gigabytes worth of data and the application would be impractical. To solve this problem the communication complexity must be sublinear in the larger set size.

Chapter 6 addresses this by building on a technique known as fully homomorphic encryption (FHE). This technique is extremely powerful in that it allows arbitrary computation to be performed on encrypted data. A party can compute the encryption of $[\![x]\!] := \mathsf{Enc}(x)$ and distribute it to anyone. This other party can then compute an arbitrary function f on the encryption $[\![x]\!]$ and obtains an encryption of the result $[\![z]\!] := f([\![x]\!])$.

Given fully homomorphic encryption it is clear that semi-honest PSI can be performed with communication proportional to the smaller set Y. First Bob computes $[\![Y]\!]$ and sending this to Alice who computes the intersection function $[\![Z]\!] := [\![Y]\!] \cap X$. This, in turn, can be sent to Bob who decrypts the result $Z = X \cap Y$. Despite this feasibility result, it has long been believed that such a computation would be incredibly slow and impractical.

Chapter 6 demonstrates that this is not the case. The primary hurdle to be overcome is to design a function computing the intersection that has constant multiplicative depth, e.g. 5. This means that the result Z must be computed using 5 or fewer consecutive multiplications. The restriction on the multiplicative depth is due to performance limitations of FHE. For simplicity, let us consider the special case where we test whether $y \in X$, i.e. $|Y| = 1$. First, consider the function:

$$z = f(y) = \prod_{x \in X} (y - x)$$

In the event that y is in fact in X, then one of these terms must be equal to zero which makes the result z equal to zero. In all other cases, z is non-zero. Bob can then conclude that $z = 0 \Leftrightarrow y \in X$. However, the multiplicative depth of this computation is $\log |X|$ when the product is computed using a tree structure. This is far too large since we assume that the size of X is on the order of 2^{28} resulting in a depth of 28.

We show how to further reduce the depth of the computation using a technique known as windowing. Observe that $f(y)$ can be rewritten as a polynomial

$$f(y) = c_N y^N + ... + c_2 y^2 + c_1 y + c_0$$

where $N = |X|$ and the coefficients c_i are a function of X. Instead of simply sending $[\![y]\!]$, Bob sends the encryptions of y raised to all the powers of two, $y^{2^0}, y^{2^1}, y^{2^2}, ..., y^{2^{\log|X|}}$. From these Alice can compute the required power of y in depth $\log\log|X|$ and complete the computation of z as a linear combination of these terms. This technique combined with several others optimizations allow the product to be computed in extremely small depth for any conceivable set X.

This technique can be extended to work with sets using the hash table technique of Pinkas et al. [PSZ14]. This in combination with the need to send $\log|X|$ powers of y bring the overall communication complexity to $O(|Y|\log|X|)$ while maintaining that the computation over fully homomorphic encryption can be done with practical performance. In particular, the intersection between $|Y| = 5000$ and $|X| = 2^{24}$ can be done in a few seconds with 12 MB of communication, roughly the size of an MP3 song. This protocol was authored by Chen, Laine & Rindal and published at the 24rd ACM Conference on Computer and Communications Security (CCS 2017) under the title "Fast Private Set Intersection from Homomorphic Encryption."

1.3.5 PSI with values from FHE

While the PSI technique of the previous section gives good performance and very low communications, many applications where such a protocol is desirable would benefit from additional functionality where key-value pairs are intersected. That is, each item is represented as a key and an associated value. The intersection is performed on the keys and for each key in the intersection, the associated value is revealed to Bob.

Consider the case of contact discovery where Bob signs up for WhatsApp. In this scenario, Bob learns which of his contacts also use WhatsApp but then must give this information over to WhatsApp, decreasing Bob's privacy to a large degree. A more privacy-preserving solution would be to use PSI with values where the associated data is Bob's public key and related meta-data. Bob could then directly establish secure communication with his contacts without using WhatsApp as a middleman.

To efficiently achieve this end a variety of techniques are introduced in Chapter 7. The main technique is to evaluate two polynomials using FHE. The first function f is the same as before which on Bob's ciphertext $[\![y]\!]$ evaluates to zero if and only if $y \in X$. For all $x \in X$, let v_x denote the corresponding value which should be returned. In addition to evaluating f, Alice evaluates a polynomial g with the property that for all $x \in X$, $g(x) = v_x$. We then have Alice return the pair of ciphertexts $f([\![y]\!]), f([\![y]\!])r + g([\![y]\!])$ where Alice samples r as a uniformly random non-zero element of the field. This pair has the property that for all $y \in X$, the pair evaluates to $(0, v_y)$. Otherwise, the pair contains uniformly random elements in the field with the first having a non-zero restriction. Using this technique combined with several other optimizations achieves the first PSI protocol capable of returning values while having sublinear communication. Moreover, this protocol also outperforms the state of the art single server Private Information Retrieval (PIR) protocol[ACLS17] while at the same time achieving stronger security guarantees. This protocol was authored by Chen, Laine, Huang & Rindal and published at the 25th ACM Conference on Computer and Communications Security (CCS 2018) under the title "Labeled PSI from Fully Homomorphic Encryption with Malicious Security."

1.3.6 Malicious Secure FHE based PSI

The question of making the FHE based PSI protocol from Section 1.3.4 malicious secure is both challenging and very interesting. We begin with an idea to get malicious security against Bob who receives output. Here the challenge is that Bob may send malformed ciphertexts which leak information about the sender's set. We propose preventing this by adding another level of encryption to the protocol.

The parties can first apply an oblivious pseudorandom function (OPRF) to their inputs. Similar to the Bloom filter and dual execution protocols, Alice samples a secret key k and Bob is allowed to learn the encodings $f_k(y)$ for all $y \in Y$. Importantly, Bob does not learn the encoding key k and Alice does not learn y. The existing PSI protocol can then be applied to the sets $\{f_k(y)\}$ and $\{f_k(x)\}$. Security against a malicious Bob follows from the OPRF. In particular, even if Bob somehow learns Alice's full set $\{f_k(x)\}$ no more information than $Y \cap X$ can be inferred due to k being unknown. Therefore no attack on the FHE protocol which takes $\{f_k(x)\}$ as input can reveal additional information. Crucial to the practicality of this approach is that the OPRF protocol can be implemented with communication only linear in the smaller set.

Malicious security against Alice is significantly more difficult to achieve. Typically malicious security requires the communication to at least be linear in the size of the malicious party's input. However, this is exactly what we wish to avoid. We opt to take a more heuristic approach. The main attack which we wish to prevent is Alice forcing Bob to output his full set simply by returning ciphertexts which encrypt zero. Recall that zero encodes that the item in question is in the intersection. The idea to prevent this attack is to utilize the PSI with values approach from above to make Alice return a special value v_x for each x. Bob includes $x = y \in Y$ in the intersection if the special value for y is returned. Intuitively the goal of this is to force Alice to know the $x = y$ to be able to compute/return v_y. We show that by placing certain requirements on how the special values are computed, it is extremely difficult to make Bob output an item not contained in the intersection $X \cap Y$. Although this approach does not prevent all attacks, it does significantly restrict the most serious attacks while not imposing a significant runtime overhead. This protocol was published in conjunction with the previous section.

1.3.7 Two-Server PIR based PSI

Chapter 8 proposes another technique to efficiently solve PSI when $|Y| \ll |X|$. This approach changes the security model so that Alice who receives no output is split into two non-colluding servers. This means that the security of this setting is conditioned on these servers not exchanging additional information beyond what the protocol specifies. While this is a strong assumption, it also enables extremely efficient PSI by employing a technique known as two-server Private Information Retrieval (PIR).

PIR allows Bob to retrieve a block of memory stored on the servers in such a way that the servers do not learn which block was retrieved. Conceptually, the PSI protocol uses PIR to first reduce Alice's set X, which is stored on the two servers, to some smaller set X' such that $|X'| \approx |Y|$. Bob and one of the server then run a traditional PSI protocol between themselves so that Bob learns $X \cap Y = X' \cap Y$.

To enable Bob to first reduce the size of X we introduce a notion called designated-output PIR where the party that learns the retrieved block need not be Bob. Instead, Bob specifies who should receive the block x which is masked as $x \oplus r$ where r is a random value chosen by Bob. The servers structure the set X as a "cuckoo hash table" such that for any $x \in X$, x will be stored at the location indexed by $h_1(x)$ or $h_2(x)$.

For all $y \in Y$, Bob performs a PIR retrieval on the locations $h_1(y)$ and $h_2(y)$ and designate one of the servers to receive the masked versions of these locations. Assuming $y \in X$, then one of these locations will equal $y \oplus r_1$ or $y \oplus r_2$. As such, Bob and this server can run a mini-PSI using the masked values as their sets. In the end, Bob learns whether $y \in X$. The primary overhead of this protocol is the PIR which requires communications of $O(\log |X|)$ per query. This results in a total communication of $O(|Y| \log |X|)$ which matches the previous approach based on fully homomorphic encryption. One advantage of this technique is that the computation the servers have to compute is much simpler and therefore the running time is reduced by a large margin. For example, an intersection between Bob with 1024 elements and Alice with 67 million elements takes 1.36 sec and uses only 4.28 MB of communication which is more than 5× faster than the fully homomorphic encryption solution. This protocol was authored by Demmler, Rindal, Rosulek & Trieu and was published at the Privacy Enhancing Technologies Symposium (PETS 2018) under the title "PIR-PSI: Scaling Private Contact discovery."

Chapter 2

Background Theory

2.0.1 Notation

We use $[n]$ to denote the set $\{1, \ldots, n\}$. Alice holds the set X and Bob Y, where $X, Y \subseteq \{0,1\}^\sigma$. A function $f : \mathbb{N} \to \mathbb{R}$ is called negligible if for every $c \in \mathbb{N}$ there exists a $N_c \in \mathbb{N}$ such that for all $x > N_c$, $|f(x)| < x^{-c}$. When describing the functionality of a protocol it is useful to distinguish the output received by each party. We use the notation $f = (f_1, f_2)$ to describe that the functionality f on input x outputs $f_1(x)$ to party 1 and $f_2(x)$ to party 2.

We use κ to denote a computational security parameter and λ to denote the statistical security parameter. These two parameters bound the probability that the adversary can violate a security property. Informally speaking, the running time of any adversary $\mathcal{A}$ with a non-negligible probability of breaking a security guarantee must require a running time of at least $O(2^\kappa)$. In addition, each execution of the protocol is permitted to fail with probability at most $O(2^{-\lambda})$. In the case of failure the adversary $\mathcal{A}$ may be able to violate any of the security properties.

2.1 Two-Party Security Models

Broadly speaking, PSI and multi-party computation in general can be divided into two security settings, *semi-honest* and *malicious*. For simplicity, we will only consider the two-party case where one of the parties is corrupt while

Parameters: σ is the bit-length of the parties' items. n is the size of the honest parties' sets. $n' > n$ is the allowed size of the maliciously corrupt party's set.

- On input $(\textsc{Receive}, \mathsf{sid}, Y)$ from Bob where $Y \subseteq \{0,1\}^\sigma$, ensure that $|Y| \leq n$ if Bob is honest, and that $|Y| \leq n'$ if Bob is corrupt. Give output $(\textsc{Bob-input}, \mathsf{sid})$ to Alice.
- Thereafter, on input $(\textsc{Send}, \mathsf{sid}, X)$ from Alice where $X \subseteq \{0,1\}^\sigma$, likewise ensure that $|X| \leq n$ if Alice is honest, and that $|X| \leq n'$ if Alice is corrupt. Give output $(\textsc{Ouput}, \mathsf{sid}, X \cap Y)$ to Bob.

Figure 2.1: Ideal functionality $\mathcal{F}_{\mathsf{PSI}}$ for private set intersection (with one-sided output)

the other is honest. The weaker and less secure setting is known as the semi-honest model where both of the participating parties faithfully follow the prescribed protocol but the corrupt party tries to infer additional information about the other party's input. This setting is sometimes called honest but curious. While this setting may seem overly weak, in many cases it can be sufficient. For instance, PSI between Google and Tesla can likely be semi-honest due to severe reputation loss if they are caught cheating. This security model also protects against accidental leakage and ensures that a passive eavesdropper cannot learn any information.

The malicious model realizes a much stronger notion of security where the corrupt party is assumed to arbitrarily deviate from the protocol specification. To prove security in this setting we must show a reduction of all possible attacks to the case where the corrupt party behave honestly but can arbitrarily choose their input. Such a reduction is called a simulator due to it simulating honest behavior given malicious behavior. Intuitively this type of proof states that all attacks are equivalent to choosing some input and therefore the same attack could occur when a trusted third party computes the function on their behalf.

In Figure 2.1 we give the ideal functionality that specifies the goal of private set intersection. We point out several facts of interest. (1) The functionality gives output only to Bob. (2) The functionality allows maliciously corrupt parties to provide larger input sets than the honest parties. This reflects that several of our protocols are unable to strictly enforce the size of an adversary's set to be the same as that of the honest party. We elaborate when discussing the security of the protocols.

2.1.1 Semi-honest Security

We use the standard notion of the static semi-honest model where one party is corrupted by the adversary at the onset of the protocol. This party follows the protocol specification exactly but will try to infer additional information by inspecting the messages that it received from the other party along with its own internal state, i.e. their input and random tape. These messages and internal state are referred to as this party's *view*.

Informally, a protocol π computing a deterministic functionality $f = (f_1, f_2)$ is semi-honest secure if the output is correct and the view of the corrupt party i only reveals information that can be inferred from their prescribed input x_i and output $f_i(x_1, x_2)$. In particular, we must show an equivalence between the **real interaction** where the protocol π is executed and an **ideal interaction** where the parties send their input to a trusted third party and receiver their output in response. In the real interaction, let the random variable $\textsc{view}_i^\pi(x_1, x_2, 1^\kappa)$ denote the view of party i when the parties execute π on inputs x_1, x_2 and security parameter κ. We say π is semi-honest secure if for each party i there exist a PPT simulator $\mathcal{S}_i$ in the ideal interaction such that

$$\{\mathcal{S}_i(1^\kappa, x_i, f_i(x_1, x_2)))\}_{x_1,x_2,\kappa} \approx \{\textsc{view}_i^\pi(x_1, x_2, 1^\kappa)\}_{x_1,x_2,\kappa}$$

where "≈" denotes computational indistinguishability. In the ideal interaction, the simulator can be viewed as a middle man between the corrupt party i and the trusted third party. The existence of $\mathcal{S}_i$ implies that the messages received by the corrupt party i can be generated only knowing the final result and as such does not leak any additional information. This setting also requires the probability that the joint output of the parties is incorrect is negligible in the security parameter.

2.1.2 Malicious Security

In the case of malicious corruptions, we define security of the PSI protocols of Chapter 3 and 4 using the standard paradigm of 2PC. In particular, our protocols are secure in the *universal composability (UC)* framework of Canetti [Can01]. Security is defined using the real/ideal, simulation-based paradigm that considers two interactions:

- In the **real interaction**, a malicious adversary $\mathcal{A}$ attacks an honest party who is running the protocol π. The honest party's inputs are cho-

Parameters: ℓ is the length of the OT strings.

- On input $(m_0, m_1) \in (\{0,1\}^\ell)^2$ from the sender and $b \in \{0,1\}$ from the receiver, give output m_b to the receiver.

Figure 2.2: Ideal functionality for 1-out-of-2 OT

sen by an *environment* $\mathcal{Z}$; the honest party also sends its final protocol output to $\mathcal{Z}$. The environment also interacts arbitrarily with the adversary. Our protocols are in a *hybrid* world, in which the protocol participants have access to an ideal random-OT functionality (Figure 2.2). We define $\textsc{real}[1^\kappa, \pi, \mathcal{Z}, \mathcal{A}]$ to be the (random variable) output of $\mathcal{Z}$ in this interaction.

- In the **ideal interaction**, a malicious adversary $\mathcal{S}$ and an honest party simply interact with the ideal functionality $\mathcal{F}$ (in our case, the ideal PSI protocol of Figure 2.1). The honest party simply forwards its input from the environment to $\mathcal{F}$ and its output from $\mathcal{F}$ to the environment. We define $\textsc{ideal}[1^\kappa, \mathcal{F}, \mathcal{Z}, \mathcal{S}]$ to be the output of $\mathcal{Z}$ in this interaction.

We say that a protocol π **UC-securely realizes** functionality $\mathcal{F}$ if: for all PPT adversaries $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$, such that for all PPT environments $\mathcal{Z}$:

$$\textsc{real}[1^\kappa, \pi, \mathcal{Z}, \mathcal{A}] \approx \textsc{ideal}[1^\kappa, \mathcal{F}, \mathcal{Z}, \mathcal{S}]$$

where "$\approx$" denotes computational indistinguishability.

2.1.3 Efficient Oblivious Transfer

Many of our protocols make use of 1-out-of-2 oblivious transfer (OT). The ideal functionality is described in Figure 2.2. We often require a large number of such OTs, secure against malicious adversaries. These can be obtained efficiently via OT extension [Bea96]. The idea is to perform a fixed number (e.g., 128) of "base OTs", and from this correlated randomness derive a large number of effective OTs using only symmetric-key primitives.

The most efficient OT extension protocols providing malicious security are those of [ALSZ15, KOS15b, OOS17], which are based on the semi-honest secure paradigm of [IKNP03].

2.1.4 Leveled Fully Homomorphic Encryption

Fully homomorphic encryption schemes are encryption schemes that allow arithmetic circuits to be evaluated directly on ciphertexts, ideally enabling powerful applications such as outsourcing of computation on private data [RAD78, Gen09, BV11, BV14, Bra12, FV12, BGV12, GSW13, CGGI16, CKKS17]. For improved performance, the encryption parameters are typically chosen to support only circuits of a certain bounded depth (*leveled* fully homomorphic encryption), and we use this in our implementations. While FHE is still far from being a generic solution to computation over encrypted data, it can be possible to achieve good performance in specific scenarios, e.g. evaluating the AES circuit [GHS12a], computing edit distance on DNA sequences [CKL15], and training logistic regression models [KSK$^+$18].

Many of the techniques and algorithms presented in Chapter 6 and 7 are agnostic to the exact fully homomorphic encryption scheme that is being used, but for simplicity we restrict to RLWE-based cryptosystems using power-of-2 cyclotomic rings of integers [LPR10]. In particular, our implementations use SEAL[KL16] for fully homomorphic encryption. In such cryptosystems the plaintext space is $\mathbb{Z}_t[x]/(x^n+1)$, and the ciphertext space is $\mathbb{Z}_q[x]/(x^n+1)$, where n is a power of 2 and $t \ll q$ are integers. It is customary to denote $R = \mathbb{Z}[x]/(x^n+1)$, so that the plaintext and ciphertext spaces become $R_t = R/tR$, and $R_q = R/qR$, respectively. We assume the fully homomorphic encryption scheme to have plaintext and ciphertext spaces of this type, and the notation (n, q, t) will always refer to these parameters. For example, the Brakerski-Gentry-Vaikuntanathan (BGV) [BGV12] and the Fan-Vercauteren (FV) [FV12] schemes have this structure.

A leveled fully homomorphic encryption scheme can be described by the following set of randomized algorithms:

- $\mathsf{FHE.Setup}(1^\kappa)$: Given a security parameter κ, outputs a set of encryption parameters $\texttt{parms}$.
- $\mathsf{FHE.KeyGen}(\texttt{parms})$: Outputs a secret key $\texttt{sk}$ and a public key $\texttt{pk}$. Optionally outputs one or more evaluation keys $\texttt{evk}$.
- $\mathsf{FHE.Encrypt}(m, \texttt{pk})$: Given message $m \in R_t$, outputs ciphertext $c \in R_q$.
- $\mathsf{FHE.Decrypt}(c, \texttt{sk})$: Given ciphertext $c \in R_q$, outputs message $m \in R_t$.

- $\mathsf{FHE.Evaluate}(C, (c_1, \ldots, c_k), \mathtt{evk})$: Given an arithmetic circuit f with k input wires, and inputs $c_1, \ldots, c_k$ with $c_i \to \mathsf{FHE.Encrypt}(m_i, \mathtt{pk})$, outputs a ciphertext c such that

  $$\Pr\left[\mathsf{FHE.Decrypt}(c, \mathtt{sk}) \neq f(m_1, \ldots, m_k)\right] = \operatorname{negl}(\kappa)\,.$$

  We also require that the size of the output of FHE.Evaluate is not more than polynomial in κ independent of what f is (*compactness*) (see e.g. [ABC+15]).

We say that a fully homomorphic encryption scheme is secure if it is IND-CPA secure, and *weakly circular secure*, which means that the scheme remains secure even when the adversary is given encryptions of the bits of the secret key. A fully homomorphic encryption scheme achieves *circuit privacy* if the distribution of the outputs of any fixed homomorphic evaluation is indistinguishable from the distribution of fresh encryptions of the plaintext outputs. In this way, one can effectively hide the circuit that was evaluated on encrypted data. We refer the reader to [ABC+15, BGV12, DS16] for more details.

The core parameters of the schemes employed are three integers: n, q, and t.[1] We set the parameters to always achieve at least a 128-bit security level according to the recommendations provided in [CCD+17]. In order to compare different parameter choices, we need to provide some rough cost estimates for basic operations in SEAL. The size of each ciphertext is $2n \log q$ bits, and the size of the underlying plaintext is $n \log t$ bits. In terms of computation, a multiplication between two ciphertexts takes $O(n \log n (\log q)^2)$ bit operations, whereas a ciphertext-plaintext multiplication takes $O(n \log n \log q)$ bit operations (see e.g. [BEHZ16]),

2.2 Related Work

2.2.1 PSI Based on Diffie-Hellman.

Private set intersection has a long and rich history of development. Dating back to 1986 Meadows [Mea86] proposed a semi-honest approach leveraging

[1] Another important parameter is the error width σ for which we use the SEAL default value 3.19.

the Diffie-Hellman key agreement protocol to generate randomized encodings of x, y such that they can securely be compared for equality. This approach was formalized by Huberman et al. [HFH99] and adapted to work with sets of items X, Y. In particular, each party samples a secret key α, β respectively. For $y \in Y$, Bob can then compute the randomized encoding $[\![y]\!] := y^{\alpha\beta}$ by first sampling a random exponent r and sending $y^{r\alpha}$ to Alice who responds with $y^{r\alpha\beta}$. Bob can then take the rth root of this to obtain $y^{\alpha\beta}$. Symmetrically, Alice can interactively compute the randomized encodings for the elements in her set. The protocol is completed by having Alice send Bob all of the encodings for her elements which allows Bob to infer $X \cap Y$ from the intersection of the encodings. Later this paradigm was extended to the malicious setting by De Cristofaro et al. [DCKT10]. While this approach achieves linear computation and communication overhead, it requires additional exponentiations over a large field which limits its practical performance.

The main benefit of this Diffie-Hellman paradigm is its low communication complexity. Indeed, protocols in this paradigm have by far the smallest communication complexity when $|X| \approx |Y|$. However, the Diffie-Hellman paradigm requires expensive public-key operations for each item in the parties' sets, making them much slower than the OT-based approaches that require only a constant number of public-key operations.

2.2.2 Unbalanced PSI Based on OPRF

As discussed in the previous section, many protocols for private set intersection are not well-suited when the two parties have input sets of very different sizes. For example, [PSZ18, KKRT16] are the fastest PSI protocols for large sets of similar size, but require communication at least $O\big(\lambda(N + n)\big)$ where N and n are the respective set sizes and λ is a statistical security parameter. This cost makes these approaches prohibitive for contact discovery, where N is very large.

In this setting Resende *et al.* [RA18] optimizes the communication overhead of PSI. The central technique of their protocol is to apply an OPRF to the receiver's set to obtain a new set $Y' = \{\text{OPRF}_k(Y) : y \in Y\}$. Here the sender holds the key k and can locally apply the OPRF to its set to obtain $X' = \{\text{OPRF}_k(x) : x \in X\}$. To reduce communication, the sender compresses the set X' before forwarding it to the receiver. While this compression does reduce the communication, it remains linear in the size of the larger set, and can introduce false positives. That is, with high compression rates the receiver outputs an element in $Y \setminus X$ with non-negligible probability.

Another work following a similar framework as [RA18] is that of Kiss *et al.* [KLS$^+$17]. The main difference between these protocols is in the choice of the OPRF and the compression technique. [RA18] uses a Diffie-Hellman based OPRF, while [KLS$^+$17] uses garbled circuits to obliviously evaluate the AES function. This alteration significantly improves the computational work required by Alice to apply the OPRF to its set at the expense of increased communication and computation when the OPRF is applied to the receiver's set. The second difference is that [KLS$^+$17] uses a more conservative compression technique and parameters, which do not introduce a significant false positive rate.

2.2.3 PSI Based on Bloom Filter

Dong, Chen & Wen [DCW13] present an approach for PSI based on representing the parties sets as Bloom filters. Their technique builds on an oblivious transfer protocol[IKNP03] and achieves practical efficiency for set size less than a hundred thousand. The Dong et al. protocol was presented in the semi-honest setting along with an extension that was claimed to have security against malicious adversaries. However, Chapter 3 identifies several bugs in their protocol and provides a provably secure solution. Similar security issues were identified by Lambæk [Lam16] but they did not propose any counter measures.

2.2.4 PSI Based on Hash Table

A more efficient approach was proposed by Pinkas, Schneider & Zohner (PSZ) [PSZ14] which combines a hash table data structure and a randomized encoding based on oblivious transfer. The encoding procedure is extremely efficient and when combined with a hash table achieves linear overhead. The PSZ approach for PSI has been improved in a series of works [PSSZ15, PSZ18, OOS17, KKRT16], with the protocol of Kolesnikov et al. [KKRT16] currently being the fastest PSI protocol against semi-honest adversaries. There have been modifications of the protocol [OOS17] that provide security against a restricted class of malicious adversaries (i.e., the protocol protects against a malicious Alice only), but besides work presented here, there has been no success in leveraging this most promising PSI paradigm to provide security in the full malicious security model.

2.2.5 PSI Based on MPC

Blanton & Aguiar[BA12] describes a relatively complete set of three-party protocols for performing intersections, unions, set difference, etc. and the corresponding SQL-like operations. These protocols could tolerate only a single corrupt party. One compelling feature is that these operations are composable in that the inputs and outputs are secret shared between the parties. At the core of their technique is the use of a general-purpose secure computation protocol and an oblivious sorting algorithm to merges the two sets followed by a linear pass over the sorted data where a relation is performed on adjacent items. This technique has the advantage of being very general and flexible. However, the proposed sorting protocol has the relatively high complexity $O(n \log^2 n)$ and is not constant round.

Huang, Evans & Katz [HEK12] discuss using general-purpose two-party secure computation (garbled circuits) to perform PSI. Their observation is that when the sets are pre-sorted the complexity of the Blanton & Aguiar protocol can be reduced to $O(n \log n)$. However, this optimization prevents the protocol from being composable. Later improvements were also suggested in [PSZ14, PSSZ15]. At the time of [HEK12], such general-purpose PSI protocols in the semi-honest setting were actually faster than other special-purpose ones. Since then, the results in OT-based PSI have made special-purpose PSI protocols significantly faster. However, we point out that using general-purpose MPC makes it relatively straight-forward to achieve security against malicious adversaries since there are many well-studied techniques for general-purpose malicious 2PC.

2.2.6 PSI Based on Oblivious Polynomial Evaluation

Another line of work was begun by Kissner and Song[KS05] and improved on by [MF06b]. Their approach is based on the observation that set intersection and multi-set union has a correspondence to operations on polynomials. A set S can be encoded as the polynomial $\hat{S}(x) = \prod_{s \in S}(x - s) \in \mathbb{F}[x]$. That is, the polynomial $\hat{S}(x)$ has a root at all $s \in S$. Given two such polynomials, $\hat{S}(x), \hat{T}(x)$, the polynomial encoding the intersection is $\hat{S}(x) + \hat{T}(x)$ with overwhelming probability given a sufficiently large field $\mathbb{F}$. Multi-set union can similarly be performed by multiplying the two polynomials together. Unlike with normal union, if an item y is contained in S and T then $\hat{S}(x)\hat{T}(x)$ will contain two roots at y which is often not the desired functionality. This general idea can be transformed into a secure multi-party protocol using oblivious polynomial evaluation[NP99] along with randomizing the resulting polynomial. The original computational overhead was $O(n^2)$ which can be reduced to the cost of polynomial interpolation $O(n \log n)$ using techniques from [MF06b]. The communication complexity is linear. In addition, this scheme assumes an ideal functionality to generate a shared Paillier key pair. We are unaware of any efficient protocol to realize this functionality except for [HMRT12] in the two-party setting.

This general approach is also composable. However, due to randomization that is performed the degree of the polynomial after each operation doubles. This limits the practical ability of the protocol to compose more than a few operations. Moreover, it is not clear how this protocol can be extended to support SQL-like queries where elements are key-value tuples.

Hazay and Nissim introduce a pair of protocols computing set intersection and union which are also based on oblivious polynomial evaluation where the roots of the polynomial encode a set. However, these protocols are restricted to the two-party case and are not composable. The non-composability comes from the fact that only one party constructs a polynomial $\hat{S}(x)$ encoding their set S while the other party obliviously evaluates it on each element in their set. The result of these evaluations is compared with zero[2]. These protocols have linear overhead and can achieve security in the malicious setting.

2.2.7 PSI Based on an Untrusted Server

Kamara et al. [KMRS14] presented techniques for both semi-honest and malicious secure PSI in a *server-aided model*. In this model, the two parties who hold data enlist the help of an untrusted third party who carries out the majority of the computation. Their protocols are extremely fast (roughly as fast as the plaintext computation) and scale to billions of input items. In our work, we primarily focus on on the more traditional (and demanding) setting where the two parties do not enlist a third party. An exception to this is made in Chapter 8

[2] The real protocol employs a slightly different comparison predicate for technical reasons.

2.2.8 PSI with Computation

Being able to compute functions on the intersection without revealing intermediate results, or even the intersection, is often a desirable property. For instance, [IKN$^+$17b] built a PSI-SUM protocol, which returns a weighted sum of all items in the intersection. This protocol is currently being used by Google to compute ad revenue conversion rates.

Pinkas *et al.* [PSWW18] presented a PSI protocol designed to perform arbitrary computation on the intersection. While their protocol is highly efficient, it has some limitations in the types of computations that can naturally be performed. Namely, a significant overhead is introduced if the function being computed is sensitive to the order of the inputs. Despite this limitation, several interesting applications were considered such as threshold-PSI, which returns true or false based on whether the intersection size reaches a certain threshold; [PSWW18, CGT12] consider PSI cardinality, which returns the size of the intersection. [NDCD$^+$13] considers a private friend-finding scenario. Ciampi and Orlandi [CO18] also design a protocol that can compute an arbitrary function on the intersection. One limitation all of these works have is that the communication complexity is at least linear in both set sizes.

2.2.9 PIR by Keywords

Private Information Retrieval (PIR) allows a user to retrieve an entry in some database held by one or more servers. A variant called *PIR by keywords* was considered by Chor *et al.* in [CGN97], where the user query consists of a keyword instead of the address of the entry in the database. Our Labeled PSI protocol can be viewed as a multi-query (single-server) PIR by keywords: we regard Bob's set Y as a set of keywords, and the set of Alice's labels as the database.

In [FIPR05] Freedman *et al.* introduced a protocol for single-server PIR by keywords based on additive homomorphic encryption. Meanwhile, using leveled FHE, our communication per keyword is $O(\sigma \log |X| + \ell)$, whereas [FIPR05] uses additive homomorphic encryption, and their communication is linear in the size of the entire database, namely $|X| \cdot \ell$. Our Labeled PSI protocol also handles multiple keywords so it can be viewed as an improvement upon their single keyword search protocol. Labeled PSI is also considered in [JL10], where the authors constructed a solution based on Diffie-Hellman type assumptions, with communication linear in the larger set.

Angel *et al.* [AS16, ACLS17] used multi-query single-server PIR by keywords as the core component of an anonymous communication protocol. In order to reduce PIR by keywords to PIR, they pushed a Bloom filter representation of the index-to-keyword map to each client. They also optimized for multi-query by using power-of-two choices and cuckoo hashing techniques.

Olumofin and Goldberg [OG10] used information-theoretical PIR by keywords to protect the privacy of client queries to a public database. Their reduction from PIR by keywords to PIR relies on B+ trees, and perfect hash functions.

2.2.10 Secure Hardware based PSI

Recently the Signal messaging service announced a solution for private contact discovery based on Intel SGX, which they plan to deploy soon [Mar17]. The idea is for the client to send their input set directly into an SGX enclave on the server, where it is privately compared to the server's set. The enclave can use remote attestation to prove the authenticity of the server software.

The security model for this approach is incomparable to ours and others, as it relies on a trusted hardware assumption. Standard two-party PSI protocols rely on standard cryptographic hardness assumptions. It is also worth pointing out that commercial uses of Intel SGX currently require a license and that there is ongoing research that focuses on applying side-channel attacks like Spectre and Meltdown to extract confidential data from SGX enclaves [Lar17].

Chapter 3

PSI From Bloom Filters

Improved Private Set Intersection against Malicious Adversaries by Peter Rindal & Mike Rosulek, in Eurocrypt[RR17a].

3.1 Introduction

There has been a great deal of recent progress in efficient PSI protocols that are secure against *semi-honest* adversaries, who are assumed to follow the protocol. The current state of the art has culminated in extremely fast PSI protocols. The fastest one, due to Kolesnikov et al. [KKRT16], can securely compute the intersection of two sets, each with 2^{20} items, in less than 4 seconds.

Looking more closely, the most efficient semi-honest protocols are those that are based on **oblivious transfer (OT) extension.** Oblivious transfer is a fundamental cryptographic primitive (see Figure 2.2). While in general OT requires expensive public-key computations, the idea of OT extension [Bea96, IKNP03] allows the parties to efficiently realize any number of *effective* OTs by using only a small number (e.g., 128) of *base OTs* plus some much more efficient symmetric-key computations. Using OT extension, oblivious transfers become extremely inexpensive in practice. Pinkas et al. [PSZ14] compared many paradigms for PSI and found the ones based on OTs are much more efficient than those based on algebraic & public-key techniques.

3.1.1 Chapter Contributions

In many settings, security against semi-honest adversaries is insufficient. *Our goal in this paper is to translate the recent success in semi-honest PSI to the setting of* ***malicious security.*** Following the discussion above, this means focusing on PSI techniques based on oblivious transfers. Indeed, recent protocols for OT extension against *malicious* adversaries [ALSZ13, KOS15b] are almost as efficient as (only a few percent more expensive than) OT extension for semi-honest adversaries.

Our starting point is the protocol paradigm of Dong, Chen & Wen [DCW13] (hereafter denoted DCW) that is based on OTs and Bloom filter encodings. We describe their approach in more detail in Section 3.2. In their work they describe one of the few malicious-secure PSI protocols based primarily on OTs rather than algebraic public-key techniques. We present the following improvements and additions to their protocol:

1. Most importantly, we show that their protocol has a subtle security flaw, which allows a malicious Alice to induce inconsistent outputs for the receiver. We present a fix for this flaw, using a very lightweight cut-and-choose technique.A

2. We present a full simulation-based security proof for the Bloom-filter-based PSI paradigm. In doing so, we identify a subtle but important aspect about using Bloom filters in a protocol meant to provide security in the presence of malicious adversaries. Namely, the simulator must be able to extract all items stored in an adversarially constructed Bloom filter. We argue that this capability is an *inherently* non-standard model assumption, in the sense that it seems to require the Bloom filter hash functions to be modeled as (non-programmable) random oracles. Details are in Section 3.4.1.

3. We implement both the original DCW protocol and our improved version. We find that the major bottleneck in the original DCW protocol is not in the cryptographic operations, but actually in a polynomial interpolation computation. The absence of polynomial interpolation in our new protocol (along with our other improvements) decreases the running time by a factor of over 8-75x.

3.1.2 Bloom Filters

A **Bloom filter (BF)** is an N-bit array B associated with k random functions $h_1, \ldots, h_k : \{0,1\}^* \to [N]$. To store an item x in the Bloom filter, one sets $B[h_i(x)] = 1$ for all i. To check the presence of an item x in the Bloom filter, one simply checks whether $B[h_i(x)] = 1$ for all i. Any item stored in the Bloom filter will therefore be detected when queried; however, *false positives* are possible.

3.2 The DCW Protocol Paradigm

The PSI protocol of Dong, Chen, and Wen [DCW13] (hereafter DCW) is based on representing the parties' input sets as Bloom filters (BFs). We describe the details of their protocol in this section.

If B and B' are BFs for two sets S and S', using the same parameters (including the same random functions), then it is true that $B \wedge B'$ (bit-wise AND) is a BF for $S \cap S'$. However, one cannot construct a PSI protocol simply by computing a bit-wise AND of Bloom filters. The reason is that $B \wedge B'$ leaks more about S and S' than their intersection $S \cap S'$. For example, consider the case where $S \cap S' = \emptyset$. Then the most natural Bloom filter for $S \cap S'$ is an all-zeroes string, and yet $B \wedge B'$ may contain a few 1s with noticeable probability. The location of these 1s depends on the items in S and S', and hence cannot be simulated just by knowing that $S \cap S' = \emptyset$.

DCW proposed a variant Bloom filter that they call a **garbled Bloom filter** (GBF). In a GBF G meant to store m-bit strings, each $G[i]$ is itself an m-bit string rather than a single bit. Then an item x is stored in G by ensuring that $x = \bigoplus_i G[h_i(x)]$. That is, the positions indexed by hashing x should store additive secret shares of x. All other positions in G are chosen uniformly.

The **semi-honest** PSI protocol of DCW uses GBFs in the following way. The two parties agree on Bloom filter parameters. Alice prepares a GBF G representing her input set. The receiver Bob prepares a standard BF B representing his input set. For each position i in the Bloom filters, the parties use oblivious transfer so that Bob can learn $G[i]$ (a string) iff $B[i] = 1$. These are exactly the positions of G that Bob needs to probe in order to determine which of his inputs is stored in G. Hence Bob can learn the intersection. DCW prove that this protocol is secure. That is, they show that Bob's view

$\{G[i] \mid B[i] = 1\}$ can be simulated given only the intersection of Alice and Bob's sets.

DCW also describe a **malicious-secure** variant of their GBF-based protocol. The main challenge is that nothing in the semi-honest protocol prevents a malicious Bob from learning *all* of Alice's GBF G. This would reveal Alice's entire input, which can only be simulated in the ideal world by Bob sending the entire universe $\{0,1\}^\sigma$ as input. Since in general the universe is exponentially large, this behavior is unsimulatable and hence constitutes an attack.

To prevent this, DCW propose to use 1-out-of-2 OTs in the following way. Bob can choose to either pick up a position $G[i]$ in Alice's GBF (if Bob has a 1 in $B[i]$) or else learn a value s_i (if Bob has a 0 in $B[i]$). The values s_i are an $N/2$-out-of-N secret sharing of some secret s^* which is used to encrypt all of the $G[i]$ values. Hence, Alice's inputs to the ith OT are $(s_i, \mathsf{Enc}(s^*, G[i]))$, where Enc is a suitable encryption scheme. Intuitively, if Bob tries to obtain too many positions of Alice's GBF (more than half), then he cannot recover the key s^* used to decrypt them.

As long as $N > 2k|Y|$ (where Y is Bob's input set), an honest Bob is guaranteed to have at least half of his BF bits set to zero. Hence, he can reconstruct s^* from the s_i shares, decrypt the $G[i]$ values, and probe these GBF positions to learn the intersection. We describe the protocol formally in Figure 3.1.

3.2.1 Insecurity of the DCW Protocol

Unfortunately, the malicious-secure variant of DCW is not secure![1] We now describe an a attack on their protocol, which was independently & concurrently discovered by Lambæk [Lam16]. A corrupt Alice will generate s_i values that are *not* a valid $N/2$-out-of-N secret sharing. DCW do not specify Bob's behavior when obtaining invalid shares. However, we argue that no matter what Bob's behavior is (e.g., to abort in this case), Alice can violate the security requirement.

As a concrete attack, let Alice honestly generate shares s_i of s^*, but then change the value of s_1 in any way. She otherwise runs the protocol as instructed. If the first bit of Bob's Bloom filter is 1, then this deviation from

[1] We contacted the authors of [DCW13], who confirmed that our attack violates malicious security.

Parameters: X is Alice's input, Y is Bob's input. N is the required Bloom filter size; We assume the parties have agreed on common BF parameters.

1. Alice chooses a random key $s^* \in \{0,1\}^\kappa$ and generates an $N/2$-out-of-N secret sharing $(s_1, \ldots, s_N)$.
2. Alice generates a GBF G encoding her inputs X. Bob generates a standard BF B encoding his inputs Y.
3. For $i \in [N]$, the parties invoke an instance of 1-out-of-2 OT, where Alice gives inputs $(s_i, c_i = \mathsf{Enc}(s^*, G[i]))$ and Bob uses choice bit $B[i]$.
4. Bob reconstructs s^* from the set of shares $\{s_i \mid B[i] = 0\}$ he obtained in the previous step. Then he uses s^* to decrypt the ciphertexts $\{c_i \mid B[i] = 1\}$, obtaining $\{G[i] \mid B[i] = 1\}$. Finally, he outputs $\{y \in Y \mid y = \bigoplus_i G[h_i(y)]\}$.

Figure 3.1: The malicious-secure protocol of DCW [DCW13].

the protocol is invisible to him, and Alice's behavior is indistinguishable from honest behavior. Otherwise, Bob will pick up s_1 which is not a valid share. If Bob aborts in this case, then his abort probability depends on whether his first BF bit is 1. The effect of this attack on Bob's output cannot be simulated in the ideal PSI functionality, so it represents a violation of security.

Even if we modify Bob's behavior to gracefully handle some limited number of invalid shares, there must be some threshold of invalid shares above which Bob (information theoretically) cannot recover the secret s^*. Whether or not Bob recovers s^* therefore depends on *individual bits* of his Bloom filter. And whether we make Bob abort or do something else (like output $\emptyset$) in the case of invalid shares, the result cannot be simulated in the ideal world. Lambæk [Lam16] points out further attacks, in which Alice can cleverly craft shares and encryptions of GBF values to cause her effective input to depend on Bob's inputs (hence violating input independence).

3.3 Our Protocol

The spirit of DCW's malicious protocol is to restrict the adversary from setting too many 1s in its Bloom filter, thereby learning too many positions in Alice's GBF. In this section, we show how to achieve the spirit of the DCW protocol using a lightweight cut-and-choose approach.

The high-level idea is to generate slightly more 1-out-of-2 OTs than the number of BF bits needed. Bob is supposed to use a limited number of 1s for his choice bits. To check this, Alice picks a small random fraction of the OTs and asks Bob to prove that an appropriate number of them used choice bit 0. If Alice uses *random* strings as her choice-bit-0 messages, then Bob can prove his choice bit by simply reporting this string.[2] If Bob cannot prove that he used sufficiently many 0s as choice bits, then Alice aborts. Otherwise, Alice has high certainty that the unopened OTs contain a limited number of choice bits 1.

After this cut-and-choose, Bob can choose a permutation that reorders the unopened OTs into his desired BF. In other words, if $c_1, \ldots, c_N$ are Bob's choice bits in the unopened OTs, Bob sends a random π such that $c_{\pi(1)}, \ldots, c_{\pi(N)}$ are the bits of his desired BF. Then Alice can send her GBF, masked by the choice-bit-1 OT messages permuted in this way.

We discuss the required parameters for the cut-and-choose below. However, we remark that the overhead is minimal. It increases the number of required OTs by only 1–10%.

3.3.1 Additional Optimizations

Starting from the basic outline just described, we also include several important optimizations. The complete protocol is described formally in Figure 3.2.

Random GBF In their treatment of the *semi-honest* DCW protocol, Pinkas et al. [PSZ14] suggested an optimization that eliminates the need for Alice to send her entire masked GBF. Suppose the parties use 1-out-of-2 OT of *random* messages (i.e., Alice does not choose the OT messages; instead, they are chosen randomly by the protocol / ideal functionality). In this case, the

[2] This *committing* property of an OT choice bit was pointed out by Rivest [Riv99].

concrete cost of OT extension is greatly reduced (cf. [ALSZ13]). Rather than generating a GBF of her inputs, Alice generates an array G where $G[i]$ is the random OT message in the ith OT corresponding to bit 1 (an honest Bob learns $G[i]$ iff the ith bit of his Bloom filter is 1).

Rather than arranging for $\bigoplus_i G[h_i(x)] = x$, as in a garbled BF, the idea is to let the G-values be random and have Alice directly send to Bob a **summary value** $K_x = \bigoplus_i G[h_i(x)]$ for each of her elements x. For each item y in Bob's input set, he can likewise compute K_y since he learned the values of G corresponding to 1s in his Bloom filter. Bob can check to see whether K_y is in the list of strings sent by Alice. For items x not stored in Bob's Bloom filter, the value K_x is random from his point of view.

Pinkas et al. show that this optimization significantly reduces the cost, since most OT extension protocols require less communication for OT of random messages. In particular, Alice's main communication now depends on the number of items in her set rather than the size of the GBF encoding her set. Although the optimization was suggested for the semi-honest variant of DCW, we point out that it also applies to the malicious variant of DCW and to our cut-and-choose protocol.

In the malicious-secure DCW protocol, the idea is to prevent Bob from seeing GBF entries unless he has enough shares to recover the key s^*. To achieve the same effect with a random-GBF, we let the choice-bit-1 OT messages be random (choice-bit-0 messages still need to be chosen messages: secret shares of s^*). These choice-bit-1 OT messages define a random GBF G for Alice. Then instead of sending a summary value $\bigoplus_i G[h_i(x)]$ for each x, Alice sends $[\bigoplus_i G[h_i(x)]] \oplus F(s^*, x)$, where F is a pseudorandom function. If Bob does not use choice-bit-0 enough, he does not learn s^* and all of these messages from Alice are pseudorandom.

In our protocol, we can let both OT messages be random, which significantly reduces the concrete overhead. The choice-bit-0 messages are used when Bob proves his choice bit in the cut-and-choose step. The choice-bit-1 messages are used as a random GBF G, and Alice sends summary values just as in the semi-honest variant.

We also point out that Pinkas et al. and DCW overlook a subtlety in how the summary values and the GBF should be constructed. Pinkas et al. specify the summary value as $\bigoplus_i G[h_i(x)]$ where h_i are the BF hash functions. Suppose that there is a collision involving two BF hash functions under the same x — that is, $h_i(x) = h_{i'}(x)$. Note that since the range of the BF hash functions is polynomial in size ($[N_{\mathsf{bf}}]$), such a collision is indeed possible with noticeable probability. When such a collision happens, the term $G[h_i(x)] = G[h_{i'}(x)]$ can cancel itself out from the XOR summation and the summary value will not depend on this term. The DCW protocol also has an analogous issue.[3] If the $G[h_i(x)]$ term was the only term unknown to the Bob, then the collision allows him to guess the summary value for an item x that he does not have. We fix this by computing the summary value using an XOR expression that eliminates the problem of colliding terms:

$$\bigoplus_{j \in h_*(x)} G[j], \qquad \text{where } h_*(x) \overset{\text{def}}{=} \{h_i(x) : i \in [k]\}.$$

Note that in the event of a collision among BF hash functions, we get $|h_*(x)| < k$.

Finally, for technical reasons, it turns out to be convenient in our protocol to define the summary value of x to be $H(x \| \bigoplus_{j \in h_*(x)} G[j])$ where H is a (non-programmable) random oracle.[4]

Hash only "on demand." In OT-extension for random messages, the parties compute the protocol outputs by taking a hash of certain values derived from the base OTs. Apart from the base OTs (whose cost is constant), these hashes account for essentially all the cryptographic operations in our protocol. We therefore modify our implementation of OT extension so that these hashes are not performed until the values are needed. In our protocol, only a small number (e.g., 1%) of the choice-bit-0 OT messages are ever used (for the cut-and-choose check), and only about half of the choice-bit-1 OT messages are needed by the sender (only the positions that would be 1 in a BF for the sender's input). Hence, the reduction in cost for the receiver is roughly 50%, and the reduction for the sender is roughly 75%. A similar optimization was also suggested by Pinkas et al. [PSZ14], since the choice-bit 0 messages are not used at all in the semi-honest protocol.

Aggregating proofs-of-choice-bits Finally, we can reduce the communication cost of the cut-and-choose step. Recall that Bob must prove that

[3] Additionally, if one strictly follows the DCW pseudocode then correctness may be violated in the event of a collision $h_i(x) = h_{i'}(x)$. If $h_i(x)$ is the first "free" GBF location then $G[h_i(x)]$ gets set to a value and then erroneously overwritten later.

[4] In practice H is instantiated with a SHA-family hash function. The XOR expression and x itself are each 128 bits, so both fit in a single SHA block.

he used choice bit 0 in a sufficient number of OTs. For the ith OT, Bob can simply send $m_{i,0}$, the random output he received from the ith OT. To prove he used choice bit 0 for an entire *set* I of indices, Bob can simply send the single value $\bigoplus_{i \in I} m_{i,0}$, rather than sending each term individually.

Optimization for programmable random oracles. The formal description of our protocol is one that is secure in the *non-programmable* random oracle model. However, the protocol can be significantly optimized by assuming a programmable random oracle. The observation is that Alice's OT input strings are always chosen randomly. Modern OT extension protocols natively give OT of random strings and achieve OT of chosen strings by sending extra correction data (cf. [ALSZ13]). If the application allows the OT extension protocol itself to determine the sender's strings, then this additional communication can be eliminated. In practice, this reduces communication cost for OTs by a factor of 2.

We can model OT of random strings by modifying the ideal functionality of Figure 2.2 to choose m_0, m_1 randomly itself. The OT extension protocol of [OOS17] securely realizes this functionality in the presence of malicious adversaries, in the programmable random oracle model. We point out that even in the semi-honest model it is not known how to efficiently realize OT of strings randomly *chosen by the functionality*, without assuming a programmable random oracle.

3.4 Security

3.4.1 BF extraction

The analysis in DCW argues for malicious security in a property-based manner, but does not use a standard simulation-based notion of security. This turns out to mask a non-trivial subtlety about how one can prove security about Bloom-filter-based protocols.

One important role of a simulator is to extract a corrupt party's input. Consider the case of simulating the effect of a corrupt Bob. In the OT-hybrid model the simulator sees Bob's OT choice bits as well as the permutation π that he sends in 5. Hence, the simulator can easily extract Bob's "effective" Bloom filter. However, the simulator actually needs to extract the receiver's *input set* that corresponds to that Bloom filter, so that it can send the set itself to the ideal functionality.

In short, the simulator must *invert* the Bloom filter. While invertible Bloom filters do exist [GM11], they require storing a significant amount of data beyond that of a standard Bloom filter. Yet this PSI protocol only allows the simulator to extract the receiver's OT choice bits, which corresponds to a *plain* Bloom filter. Besides that, in our setting we must invert a Bloom filter that may not have been honestly generated.

Our protocol achieves extraction by modeling the Bloom filter hash functions as (non-programmable) random oracles. The simulator must *observe* the adversary's queries to the Bloom filter hash functions.[5] Let Q be the set of queries made by the adversary to any such hash function. This set has polynomial size, so the simulator can probe the extracted Bloom filter to test each $q \in Q$ for membership. The simulator can take the appropriate subset of Q as the adversary's extracted input set. More details are given in the security proof below.

Simulation/extraction of a corrupt Alice is also facilitated by observing her oracle queries. Recall that the *summary value* of x is (supposed to be) $H(x\| \bigoplus_{j \in h_*(x)} m_{\pi(j),1})$. Since H is a non-programmable random oracle, the simulator can obtain candidate x values from her calls to H.

More details about malicious Bloom filter extraction are given in the security proof in Section 8.4.

Necessity of random oracles. We show that random oracles are necessary, when using plain Bloom filters for a PSI protocol.

Lemma 1. *There is* ***no*** *PSI protocol that simultaneously satisfies the following conditions:*

- *The protocol is UC secure against malicious adversaries in the standard model.*
- *When Bob is corrupted in a semi-honest manner, the view of the simulator can be sampled given only on a Bloom filter representation of Bob's input.*

[5] The simulator does not, however, require the ability to *program* the random oracle.

- *The parameters of the Bloom filter depend only on the number of items in the parties' sets, and in particular not on the bitlength of those items.*

In our protocol, the simulator's indeed gets to see the receiver's OT choice bits, which correspond to a plain Bloom filter encoding of their input set. However, the simulator also gets to observe the receiver's random oracle queries, and hence the statement of the lemma does not apply.

The restriction about the Bloom filter parameters is natural. One important benefit of Bloom filters is that they do not depend on the bit-length of the items being stored.

Proof. Consider an environment that chooses a random set $S \subseteq \{0,1\}^\ell$ of size n, and gives it as input to both parties (ℓ will be chosen later). An adversary corrupts Bob but runs semi-honestly on input S as instructed. The environment outputs 1 if the output of the protocol is S (note that it does not matter if only one party receives output). In this real execution, the environment outputs 1 with overwhelming probability due to the correctness of the protocol.

We will show that if the protocol satisfies all three conditions in the lemma statement, then the environment will output 0 with constant probability in the ideal execution, and hence the protocol will be insecure.

Suppose the simulator for a corrupt Bob sees only a Bloom filter representation of Bob's inputs. Let N be the total length of the Bloom filter representation (the Bloom filter array itself as well as the description of hash functions). Set the length of the input items $\ell > 2N$. Now the simulator's view can be sampled given only N bits of information about S, whereas S contains randomly chosen items of length $\ell > 2N$. The simulator must extract a value S' and send it on behalf of Bob to the ideal functionality. With constant probability this S' will fail to include some item of S (it will likely not include any of them). Then since the honest party gave input S, the output of the functionality will be $S \cap S' \neq S$, and the environment outputs zero. □

3.4.2 Cut-and-choose parameters

The protocol mentions various parameters:

N_{ot}: the number of OTs

N_{bf}: the number of Bloom filter bits

k: the number of Bloom filter hash functions

α: the fraction of 1s among Bob's choice bits

p_{chk}: the fraction of OTs to check

N_{maxones}: the maximum number of 1 choice bits allowed to pass the cut-and-choose.

As before, we let κ denote the computational security parameter and λ denote the statistical security parameter.

We require the parameters to be chosen subject to the following constraints:

- *The cut-and-choose restricts Bob to few 1s.* Let N_1 denote the number of OTs that remain after the cut and choose, in which Bob used choice bit 1. In the security proof we argue that the difficulty of finding an element stored in the Bloom filter *after the fact* is $(N_1/N)^k$ (i.e., one must find a value which all k random Bloom filter hash functions map to a 1 in the BF).

  Let $\mathcal{B}$ denote the "bad event" that no more than N_{maxones} of the checked OTs used choice bit one (so Bob can pass the cut-and-choose), and yet $(N_1/N_{\mathsf{bf}})^k \geq 2^{-\kappa}$. We require $\Pr[\mathcal{B}] \leq 2^{-\lambda}$.

  As mentioned above, the spirit of the protocol is to restrict a corrupt receiver from setting too many 1s in its (plain) Bloom filter. DCW suggest to restrict the receiver to 50% 1s, but do not explore how the fraction of 1s affects security (except to point out that 100% 1s is problematic). Our analysis pinpoints precisely how the fraction of 1s affects security.

- *The cut-and-choose leaves enough OTs unopened for the Bloom filter.* That is, when choosing from among N_{ot} items, each with independent p_{chk} probability, the probability that less than N_{bf} remain unchosen is at most $2^{-\lambda}$.

- *The honest Bob has enough one choice bits after the cut and choose.* When inserting n items into the bloom filter, at most nk bits will be set to one. We therefore require that no fewer than this remain after the cut and choose.

Our main technique is to apply the Chernoff bound to the probability that Bob has too many 1s after the cut and choose. Let $m_h^1 = \alpha N_{\mathsf{ot}}$ (resp. $m_h^0 = (1-\alpha)N_{\mathsf{ot}}$) be the number of 1s (resp. 0s) Bob is supposed to select in the OT extension. Then in expectation, there should be $m_h^1 p_{\mathsf{chk}}$ ones in the cut and choose open set, where each OT message is opened with independent probability p_{chk}. Let ϕ denote the number of ones in the open set. Then applying the Chernoff bound we obtain,

$$\Pr[\phi \geq (1+\delta)m_h^1 p_{\mathsf{chk}}] \leq e^{-\frac{\delta^2}{2+\delta}m_h^1 p_{\mathsf{chk}}} \leq 2^{-\lambda}$$

where the last step bounds this probability to be negligible in the statistical security parameter λ. Solving for δ results in,

$$\delta \leq \frac{\lambda + \sqrt{\lambda^2 + 8\lambda m_h^1 p_{\mathsf{chk}}}}{2m_h^1 p_{\mathsf{chk}}}.$$

Therefore an honest Bob should have no more than $N_{\mathsf{maxones}} = (1+\delta)m_h^1 p_{\mathsf{chk}}$ 1s revealed in the cut and choose, except with negligible probability. To ensure there are at least nk ones[6] remaining to construct the bloom filter, set $m_h^1 = nk + N_{\mathsf{maxones}}$. Similarly, there must be at least N_{bf} unopened OTs which defines the total number of OTs to be $N_{\mathsf{ot}} = N_{\mathsf{bf}} + (1+\delta^*)N_{\mathsf{ot}}p_{\mathsf{chk}}$ where δ^* is analogous to δ except with respect to the total number of OTs opened in the cut and choose.

A malicious Bob can instead select $m_a^1 \geq m_h^1$ ones in the OT extension. In addition to Bob possibly setting more 1s in the BF, such a strategy will increase the probability of the cut and choose revealing more than N_{maxones} 1s. A Chernoff bound can then be applied to the probability of seeing a δ' factor fewer 1s than expected. Bounding this to be negligible in the statistical security parameter λ, we obtain,

$$\Pr[\phi \leq (1-\delta')p_{\mathsf{chk}}m_a^1] \leq e^{-\frac{\delta'^2}{2}p_{\mathsf{chk}}m_a^1} \leq 2^{-\lambda}.$$

Solving for δ' then yields $\delta' \leq \sqrt{\frac{2\lambda}{p_{\mathsf{chk}}m_a^1}}$. By setting N_{maxones} equal to $(1-\delta')p_{\mathsf{chk}}m_a^1$ we can solve for m_a^1 such that the intersection of these two distribution is negligible. Therefore the maximum number of 1s remaining is $N_1 = (1-p_{\mathsf{chk}})m_a^1 + \sqrt{2\lambda p_{\mathsf{chk}}m_a^1}$.

For a given p_{chk}, n, k, the above analysis allows us to bound the maximum

[6] nk ones is an upper bound on the number of ones required. A tighter analysis could be obtained if collisions were accounted for.

advantage a malicious Bob can have. In particularly, a honest Bob will have at least nk 1s and enough 0s to construct the bloom filter while a malicious Bob can set no more than $N_1/N_{\textsf{bf}}$ fraction of bits in the bloom filter to 1. Modeling the bloom filter hash function as random functions, the probability that all k index the boom filter one bits is $(N_1/N_{\textsf{bf}})^k$. Setting this to be negligible in the computational security parameter κ we can solve for $N_{\textsf{bf}}$ given N_1 and k. The overall cost is therefore $\frac{N_{\textsf{bf}}}{(1-p_{\textsf{chk}})}$. By iterating over values of k and $p_{\textsf{chk}}$ we obtain set of parameters shown in Figure 3.1.

3.4.3 Security Proof

Theorem 2. *The protocol in Figure 3.2 is a UC-secure protocol for PSI in the random-OT-hybrid model, when H and the Bloom filter hash functions are non-programmable random oracles, and the other protocol parameters are chosen as described above.*

Proof. We first discuss the case of a corrupt receiver Bob, which is the more difficult case since we must not only extract Bob's input but simulate the output. The simulator behaves as follows:

> The simulator plays the role of an honest Alice and ideal functionalities in steps 1 through 5, but also extracts all of Bob's choice bits b for the OTs. Let N_1 be the number of OTs with choice bit 1 that remain after the cut and choose. The simulator artificially aborts if Bob succeeds at the cut and choose and yet $(N_1/N_{\textsf{bf}})^k \geq 2^{-\kappa}$. From the choice of parameters, this event happens with probability only $2^{-\lambda}$.
>
> After receiving Bob's permutation π in step 5, the simulator computes Bob's effective Bloom filter $BF[i] = b_{\pi(i)}$. Let Q be the set of queries made by Bob to *any* of the Bloom filter hash functions (random oracles). The simulator computes $\tilde{Y} = \{q \in Q \mid \forall i : BF[h_i(q)] = 1\}$ as Bob's effective input, and sends $\tilde{Y}$ to the ideal functionality. The simulator receives $Z = X \cap \tilde{Y}$ as output, as well as $|X|$. For $z \in Z$, the simulator generates $K_z = H(z \parallel \bigoplus_{j \in h_*(z)} m_{\pi(j),1})$. The simulator sends a random permutation of K_z along with $|X| - |Z|$ random strings to simulate Alice's message in step 6.

To show the soundness of this simulation, we proceed in the following sequence of hybrids:

1. The first hybrid is the real world interaction. Here, an honest Alice also queries the random oracles on her actual inputs $x \in X$. For simplicity later on, assume that Alice queries her random oracle as late as possible (in step 6 only).

2. In the next hybrid, we artifically abort in the event that $(N_1/N_{\mathsf{bf}})^k \geq 2^{-\kappa}$. As described above, our choice of parameters ensures that this abort happens with probability at most $2^{-\lambda}$, so the hybrids are indistinguishable.

   In this hybrid, we also observe Bob's OT choice bits. Then in step 5 of the protocol, we compute Q, BF, and $\tilde{Y}$ as in the simulator description above.

3. We next consider a sequence of hybrids, one for each item x of Alice such that $x \in X \setminus \tilde{Y}$. In each hybrid, we replace the summary value $K_x = H(x \parallel \bigoplus_{j \in h_*(x)} m_{\pi(j),1})$ with a uniformly random value.

   There are two cases for $x \in X \setminus \tilde{Y}$:

   - Bob queried some h_i on x before step 5: If this happened but x was not included in $\tilde{Y}$, then x is *not* represented in Bob's effective Bloom filter BF. There must be an i such that Bob did not learn $m_{\pi(h_i(x)),1}$.
   - Bob did *not* query any h_i on x: Then the value of $h_i(x)$ is random for all i. The probability that x is present in BF is the probability that $BF[h_i(x)] = 1$ for all i, which is $(N_1/N_{\mathsf{bf}})^k$ since Bob's effective Bloom filter has N_1 ones. Recall that the interaction is already conditioned on the event that $(N_1/N_{\mathsf{bf}})^k < 2^{-\kappa}$. Hence it is with overwhelming probability that Bob did not learn $m_{\pi(h_i(x)),1}$ for some i.

   In either case, there is an i such that Bob did not learn $m_{\pi(h_i(x)),1}$, so that value is random from Bob's view. Then the corresponding sum $\bigoplus_{j \in h_*(x)} m_{\pi(j),1}$ is uniform in Bob's view.[7] It is only with negligible probability that Bob makes the oracle query $K_x = H(x \parallel \bigoplus_{j \in h_*(x)} m_{\pi(j),1})$. Hence K_x is pseudorandom and the hybrids are indistinguishable.

[7] This is part of the proof that breaks down if we compute a summary value using $\bigoplus_i m_{\pi(h_i(x)),1}$ instead of $\bigoplus_{j \in h_*(x)} m_{\pi(j),1}$. In the first expression, it may be that $h_{i'}(x) = h_i(x)$ for some $i' \neq i$ so that the randomizing term $m_{\pi(h_i(x)),1}$ cancels out in the sum.

In the final hybrid, the simulation does not need to know X, it only needs to know $X \cap \tilde{Y}$. In particular, the values $\{K_x \mid x \in X \setminus \tilde{Y}\}$ are now being simulated as random strings. The interaction therefore describes the behavior of our simulator interacting with corrupt Bob.

Now consider a corrupt Alice. The simulation is as follows:

> The simulator plays the role of an honest Bob and ideal functionalities in steps 1 through 4. As such, the simulator knows Alice's OT messages $m_{i,b}$ for all i, b, and can compute the correct r^* value in step 4. The simulator sends a completely random permutation π in step 5.
>
> In step 6, the simulator obtains a set K as Alice's protocol message. Recall that each call made to random oracle H has the form $q \| s$. The simulator computes $Q = \{q \mid \exists s : \text{ Alice queried } H \text{ on } q \| s\}$. The simulator computes $\tilde{X} = \{q \in Q \mid H(q \parallel \bigoplus_{j \in h_*(q)} m_{\pi(j),1}) \in K\}$ and sends $\tilde{X}$ to the ideal functionality as Alice's effective input. Recall Alice receives no output.

It is straight-forward to see that Bob's protocol messages in steps 4 & 5 are distributed independently of his input.

Recall that Bob outputs $\{y \in Y \mid H(y \parallel \bigoplus_{j \in h_*(y)} m^*_{\pi(j)}) \in K\}$ in the last step of the protocol. In the ideal world (interacting with our simulator), Bob's output from the functionality is $\tilde{X} \cap Y = \{y \in Y \mid y \in \tilde{X}\}$. We will show that the two conditions are the same except with negligible probability. This will complete the proof.

We consider two cases:

- If $y \in \tilde{X}$, then $H(y \parallel \bigoplus_{j \in h_*(y)} m^*_{\pi(j)}) = H(y \parallel \bigoplus_{j \in h_*(y)} m_{\pi(j),1}) \in K$ by definition.
- If $y \notin \tilde{X}$, then Alice never queried the oracle $H(y \| \cdot)$ before fixing K, hence $H(y \parallel \bigoplus_{j \in h_*(y)} m^*_{\pi(j)})$ is a fresh oracle query, distributed independently of K. The output of this query appears in K with probability $|K|/2^\kappa$.

Taking a union bound over $y \in Y$, we have that, except with probability $|K||Y|/2^\kappa$,

$$H(y \parallel \textstyle\bigoplus_{j \in h_*(y)} m^*_{\pi(j)}) \in K \iff y \in \tilde{X}$$

Hence Bob's ideal and real outputs coincide. □

Size of the adversary's input set. When Alice is corrupt, the simulator extracts a set $\tilde{X}$. Unless the adversary has found a collision under random oracle H (which is negligibly likely), we have that $|\tilde{X}| \leq |K|$. Thus the protocol enforces a straightforward upper bound on the size of a corrupt Alice's input.

The same is not true for a corrupt Bob. The protocol enforces an upper bound only on the size on Bob's *effective Bloom filter* and a bound on the number of 1s in that BF. We now translate these bounds to derive a bound on the size of the set extracted by the simulator. Note that the ideal functionality for PSI (Figure 2.1) explicitly allows corrupt parties to provide larger input sets than honest parties.

First, observe that only queries made by the adversary before step 5 of the protocol are relevant. Queries made by the adversary *after* do not affect the simulator's extraction. As in the proof, let Q be the set of queries made by Bob before step 5. Bob is able to construct a BF with at most N_1 ones, and causing the simulator to extract items $\tilde{Y} \subseteq Q$, only if:

$$\left| \bigcup_{y \in \tilde{Y}; i \in [k]} h_i(y) \right| \leq N_1.$$

Then by a union bound over all Bloom filters with N_1 bits set to 1, and all $\tilde{Y} \subseteq Q$ of size $|\tilde{Y}| = n'$, we have:

$$\Pr\left[\begin{array}{c}\text{simulator extracts}\\ \text{some set of size } n'\end{array}\right] \leq \binom{|Q|}{n'}\binom{N_{\mathsf{bf}}}{N_1}\left(\frac{N_1}{N_{\mathsf{bf}}}\right)^{kn'}.$$

The security proof already conditions on the event that $(N_1/N_{\mathsf{bf}})^k \leq 2^{-\kappa}$, so we get:

$$\begin{aligned}\Pr\left[\begin{array}{c}\text{simulator extracts}\\ \text{some set of size } n'\end{array}\right] &\leq \binom{|Q|}{n'}\binom{N_{\mathsf{bf}}}{N_1} 2^{-\kappa n'} \\ &\leq \left(|Q|^{n'}\right)\left(2^{N_{\mathsf{bf}}}\right) 2^{-\kappa n'}\end{aligned}$$

To make the probability less than $2^{-\kappa}$ it therefore suffices to have $n' = (\kappa + N_{\mathsf{bf}})/(\kappa - \log|Q|)$.

In our instantiations, we always have $N_{\mathsf{bf}} \leq 3\kappa n$, where n denotes the *intended* size of the parties' sets. Even in the pessimistic case that the adversary makes $|Q| = 2^{\kappa/2}$ queries to the Bloom filter hash functions, we have $n' \approx 6n$. Hence, the adversary is highly unlikely to produce a Bloom filter containing 6 times the intended number of items. We emphasize that this is a very loose bound, but show it just to demonstrate that the simulator indeed extracts from the adversary a modestly sized effective input set.

3.4.4 Non-Programmable Random Oracles in the UC Model

Our protocol makes significant use of a non-programmable random oracle. In the standard UC framework [Can01], the random oracle must be treated as *local* to each execution for technical reasons. The UC framework does not deal with global objects like a single random oracle that is used by many protocols/instances. Hence, as currently written, our proof implies security when instantiated with a highly local random oracle.

Canetti, Jain, & Scafuro [CJS14] proposed a way to model global random oracles in the UC framework (we refer to their model as UC-gRO). One of the main challenges is that (in the plain UC model) the simulator can observe the adversary's oracle queries, but an adversary can ask the environment to query the oracle on its behalf, hidden from the simulator. In the UC model, every functionality and party in the UC model is associated with a *session id* (sid) for the protocol instance in which it participates. The idea behind UC-gRO is as follows:

- There is a functionality gRO that implements an ideal random oracle. Furthermore, this functionality is **global** in the sense that all parties and all functionalities can query it.
- Every oracle query in the system must be prefixed with some sid.
- There is no enforcement that oracle queries are made with the "correct" sid. Rather, if a party queries gRO with a sid that does not match its own, that query is marked as **illegitimate** by gRO.

- A functionality can ask gRO for all of the illegitimate queries made using that functionality's sid.

Our protocol and proof can be modified in the following ways to provide security in the UC-gRO model:

1. In the protocol, all queries to relevant random oracles (Bloom filter functions h_i and outer hash function H) are prefixed with the sid of this instance.
2. The ideal PSI functionality is augmented in a standard way of UC-gRO: When the adversary/simulator gives the functionality a special command `illegitimate`, the functionality requests the list of illegitimate queries from gRO and forwards them to the adversary/simulator.
3. In the proof, whenever the simulator is described as obtaining a list of the adversary's oracle queries, this is done by observing the adversary's queries and also obtaining the illegitimate queries via the new mechanism.

With these modifications, our proof demonstrates security in the UC-gRO model.

3.5 Performance Evaluation

We implemented our protocol in addition to the protocols of DCW [DCW13] outlined in Section 3.2 and that of DKT [DCKT10]. All source code can be found at https://github.com/osu-crypto/libPSI. In this section we report on their performance and analyze potential trade offs.

3.5.1 Implementation & Test Platform

In the offline phase, our protocol consists of performing 128 base OTs using the protocol of [NP01]. We extend these base OTs to N_{ot} OTs using an optimized implementation of the Keller et al. [KOS15b] OT extension protocol. Our implementation uses the programmable-random-oracle optimization for OT of random strings, described in Section 8.12. In the multi-threaded case, the OT extension and Base OTs are performed in parallel. Subsequently, the cut and choose seed is published which determines the set of OT messages to be opened. Then one or more threads reports the choice bits used for the corresponding OT and the XOR sum of the messages. Alice validates the reported value and proceeds to the online phase.

The online phase begins with both parties inserting items into a plaintext bloom filter using one or more threads. As described in section 3.4.1, the BF hash functions should be modeled as (non-programmable) random oracles. We use SHA1 as a random oracle but then expand it to a suitable length via a fast PRG (AES in counter mode) to obtain:[8]

$$h_1(x) \| h_2(x) \| \cdots \| h_k(x) = \mathsf{PRG}(\mathsf{SHA1}(x)).$$

Hence we use just one (slow) call to SHA to compute all BF hash functions for a single element, which significantly reduces the time for generating Bloom filters. Upon the computing the plaintext bloom filter, the receiver selects a random permutation mapping the random OT choice bits to the desired bloom filter. The permutation is published and Alice responds with the random garbled bloom filter masks which correspond to their inputs. Finally, the receiver performs a plaintext intersection of the masks and outputs the corresponding values.

We evaluated the prototype on a single server with simulated network latency and bandwidth. The server has 2 36-cores Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz and 256GB of RAM (e.i. 36 cores & 128 GB per party). We executed our prototype in two network settings: a LAN configuration with both parties in the same network with 0.2 ms round-trip latency, 1 Gbps; and a WAN configuration with a simulated 95 ms round-trip latency, 60 Mbps. All experiments we performed with a computational security parameter of $\kappa = 128$ and statistical security parameter $\lambda = 40$. The times reported are an average over 10 trials. The variance of the trials was between $0.1\% - 5.0\%$ in the LAN setting and $0.5\% - 10\%$ in the WAN setting with a trend of smaller variance as n becomes larger. The CPUs used in the trials had AES-NI instruction set for fast AES computations.

[8] Note that if we model SHA1 as having its queries observable to the simulator, then this property is inherited also when expanding the SHA1 output with a PRG.

3.5.2 Parameters

We demonstrate the scalability of our implementation by evaluating a range of set sizes $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$ for strings of length $\sigma = 128$. In all of our tests, we use system parameters specified in Figure 3.1. The parameters are computed using the analysis specified in Section 3.4.2. Most importantly they satisfy that except with probability negligible in the computation security parameter κ, a receiver after step 5 of Figure 3.2 will not find an x not previously queried which is contained in the garbled bloom filter.

The parameters are additionally optimized to reduce the overall cost of the protocol. In particular, the total number of OTs $N_{\mathsf{ot}} = N_{\mathsf{bf}}/(1 - p_{\mathsf{chk}})$ is minimized. This value is derived by iterating over all the region of $80 \leq k \leq 100$ hash functions and cut-and-choose probabilities $0.001 \leq p_{\mathsf{chk}} \leq 0.1$. For a given value of n, k, p_{chk}, the maximum number of ones N_1 which a possibly malicious receiver can have after the cut and choose is defined as shown in Section 3.4.2. This in turn determines the minimum value of N_{bf} such that $(N_{\mathsf{bf}}/N_1)^{-k} \leq 2^{-\kappa}$ and therefore the overall cost N_{ot}. We note that for κ other than 128, a different range for the number of hash functions should be considered.

n	p_{chk}	k	N_{ot}	N_{bf}	α	N_{maxones}
2^8	0.099	94	99,372	88,627	0.274	3,182
2^{12}	0.053	94	1,187,141	1,121,959	0.344	22,958
2^{16}	0.024	91	16,992,857	16,579,297	0.360	150,181
2^{20}	0.010	90	260,252,093	257,635,123	0.366	962,092

Table 3.1: Optimal Bloom filter cut and choose parameters for set size n to achieve statistical security $\lambda = 40$ and computational security $\kappa = 128$. N_{ot} denotes the total number of OTs used. N_{bf} denotes the bit count of the bloom filer. α is the faction of ones which should be generated. N_{maxones} is the maximum number of ones in the cut and choose to pass.

3.5.3 Comparison to Other Protocols

For comparison, we implemented two other protocol paradigms, which we describe here:

Setting	Protocol	Set size n							
		2^8		2^{12}		2^{16}		2^{20}	
		Total	Online	Total	Online	Total	Online	Total	Online
LAN	*DCW (Fig. 3.1)	3.0	(1.4)	58.5	(27.8)	1,134	(532)	-	
	*DCW + RGBF	2.9	(1.4)	58.4	(27.6)	1,145	(542)	-	
	DKT	1.7		22.6		358		3,050	
	Ours (Fig 3.2)	**0.2**	(0.003)	**0.9**	(0.04)	**9.7**	(0.7)	**127**	(14)
WAN	*DCW (Fig. 3.1)	4.2	(1.8)	61.3	(28.8)	1,185	(532)	-	
	*DCW + RGBF	4.0	(1.6)	60.6	(28.6)	1,189	(530)	-	
	DKT	1.7		23.1		393		5,721	
	Ours (Fig 3.2)	**0.95**	(0.1)	**4.6**	(0.8)	**56**	(11)	**935**	(175)

Table 3.2: Total time in seconds, with online time in parentheses, for PSI of two sets of size n with elements of 128 bits. The LAN (resp. WAN) setting has 0.2ms (resp. 95ms) round trip time latency. As noted in Section 3.5.3, when the protocol is marked with an asterisk, we report an optimistic underestimate of the running time. Missing times (-) took > 5 hours.

Threads	Protocol	Set size n			
		2^8	2^{12}	2^{16}	2^{20}
4	DKT	0.79	6.75	98.1	1,558
	Ours (Fig 3.2)	0.17	0.63	4.3	66
16	DKT	0.36	2.56	31.0	461
	Ours (Fig 3.2)	0.17	0.46	3.8	51
64	DKT	0.17	1.30	20.1	309
	Ours (Fig 3.2)	0.17	0.30	2.3	37

Table 3.3: Total running time in seconds for the DKT and our protocol when 4, 16, and 64 threads per party are used. The evaluations were performed in the LAN setting with a 0.2ms round trip time.

	set size n				asymptotic	
	2^8	2^{12}	2^{16}	2^{20}	Offline	Online
DCW (Fig. 3.1)	3.2	50.7	810	-	$2n\kappa^2$	$4nk^2$
DCW + RGBF	2.4	33.9	541	-	$2n\kappa^2$	$2n\kappa^2 + n\kappa$
DKT	0.05	0.8	14	213	0	$6n\phi + 6\phi + n\kappa$
Ours (Fig 3.2)	1.9	23	324	4,970	$2n\kappa^2$	$2n\kappa \log_2(2n\kappa) + n\kappa$

Table 3.4: The empirical and asymptotic communication cost for sets of size n reported in megabytes, and bits respectively. $\phi = 283$ is the size of the elliptic curve elements. Missing entries had prohibitively long running times and are estimated to be greater than $8,500$MB.

DCW protocol. Our first point of comparison is to the protocol of Dong, Chen, & Wen [DCW13], on which ours is based. The protocol is described in Section 3.2. While their protocol has issues with its security, our goal here is to illustrate that our protocol also has significantly better performance.

In [DCW13], the authors implement only their semi-honest protocol variant, not the malicious one. An aspect of the malicious DCW protocol that is easy to overlook is its reliance on an $N/2$-out-of-N secret sharing scheme. When implementing the protocol, it becomes immediately clear that such a secret-sharing scheme is a major computational bottleneck.

Recall that Alice generates shares from such a secret sharing scheme, and the receiver reconstructs such shares. In this protocol, the required N is the number of bits in the Bloom filter. As a concrete example, for PSI of sets of size 2^{20}, the Bloom filter in the DCW protocol has roughly 2^{28} bits. Using Shamir secret sharing, Alice must evaluate a random polynomial of degree $\sim 2^{27}$ on $\sim 2^{28}$ points. Alice must interpolate such a polynomial on $\sim 2^{27}$ points to recover the secret. Note that the polynomial will be over $GF(2^{128})$, since the protocol secret-shares an (AES) encryption key.

We chose not to develop a full implementation of the malicious DCW protocol. Rather, we fully implemented the [garbled] Bloom filter encoding steps and the OTs. We then **simulated** the secret-sharing and reconstruction steps in the following way. We calculated the number of field multiplications that would be required to evaluate a polynomial of the suitable degree by the Fast Fourier Transform (FFT) method, and simply had each party perform the appropriate number of field multiplications in $GF(2^{128})$. The field was instantiated using the NTL library with all available optimizations enabled. Our simulation significantly underestimates the cost of secret sharing in the DCW protocol, since: (1) it doesn't account for the cost associated with virtual memory accesses when computing on such a large polynomial; and (2) evaluating/interpolating the polynomial via FFT reflects a *best-case scenario*, when the points of evaluation are roots of unity. In the protocol, the receiver Bob in particular does not have full control over which points of the polynomial he will learn.

Despite this optimistic simulation of the secret-sharing step, its cost is substantial, accounting for 97% of the execution time. In particular, when comparing our protocol to the DCW protocol, the main difference in the online phase is the secret sharing reconstruction which accounts for a 113× increase in the online running time for $n = 2^{16}$.

We simulated two variants of the DCW malicious-secure protocol. One variant reflects the DCW protocol as written, using OTs of chosen messages. The other variant includes the "random GBF" optimization inspired by [PSZ14] and described in Section 3.3. In this variant, one of the two OT messages is set randomly by the protocol itself, and not chosen by Alice. This reduces the online communication cost of the OTs by roughly half. However, it surprisingly has a slight negative effect on total time. The reason is that during the online phase Alice has more than enough time to construct and send a plain GBF while Bob performs the more time intensive secret-share reconstruction step. For $n = 2^{16}$, the garbled bloom filter takes less than 5% of the secret share reconstruction time to be sent. When using a randomized GBF, Alice sends summary values to Bob, which he must compare to his own summary values. Note that there is a summary value for each item in a party's set (e.g., 2^{20}), so these comparisons involve lookups in some non-trivial data structure. This extra computational effort is part of the the critical path since the Bob has to do it. In summary, the "random GBF" optimization does reduce the required communication, however it also increases the critical path of the protocol due to the secret-share reconstruction hiding the effects of this communication savings and the small additional overhead of performing n lookups.

DH-based PSI protocols. Another paradigm for PSI uses public-key techniques and is based on Diffie-Hellman-type assumptions in cyclic groups. The most relevant protocol in this paradigm that achieves malicious security is that of De Cristofaro, Kim, and Tsudik [DCKT10] which we refer to as DKT. While protocols in this paradigm have extremely low communication complexity, they involve a large number of computationally expensive public-key operations (exponentiations). Another potential advantage of the DKT protocol over schemes based on Bloom filters is that the receiver can be restricted to a set size of exactly n items. This is contrasted with our protocol where the receiver can have a set size of $n' \approx 6n$.

We fully implemented the [DCKT10] PSI protocol both in the single and multi threaded setting. In this protocol, the parties perform $5n$ exponentiations and $2n$ related zero knowledge proofs of discrete log equality. Following the suggestions in [DCKT10], we instantiate the zero knowledge proofs in the RO model with the Fiat-Shamir transform applied to a sigma protocol. The resulting PSI protocol has in total $12n$ exponentiations along with several other less expensive group operations. The implementation is built on the Miracl elliptic curve library using Curve 25519 achieving 128 bit computational security. The implementation also takes advantage of the Comb method to perform a precomputation to increase the speed of exponentiations (point multiplication). Additionally, all operations are performed in a streaming manner allowing for the greatest amount of work to be performed concurrently by the parties.

3.5.4 Results

The running time of our implementation is shown in Figure 3.2. We make the distinction of reporting the running times for both the total time and online phase when applicable. The offline phase contains all operations which are independent of the input sets. For the bloom filter based protocols the offline phase consists of performing the OT extension and the cut and choose. Out of these operations, the most time-consuming is the OT extension. For instance, with $n = 2^{20}$ we require 260 million OTs which requires 124 seconds; the cut and choose takes only 3 seconds. For the smaller set size of $n = 2^{12}$, the OT extension required $461ms$ and the cut and choose completed in $419ms$. The relative increase in the cut and choose running time is primarily due to the need to open a larger portion of the OTs when n is smaller.

The online phase consists of the receiver first computing their bloom filter. For set size $n = 2^{20}$, computing the bloom filter takes 6.4 seconds. The permutation mapping the receiver's OTs to the bloom filter then computed in less than a second and sent. Upon receiving the permutation, Alice computes their PSI summary values and sends them to the receiver. This process when $n = 2^{20}$ takes roughly 6 seconds. The receiver then outputs the intersection in less than a second.

As expected, our optimized protocol achieves the fastest running times compared to the other malicious secure constructions. When evaluating our implementation with a set size of $n = 2^8$ on a single thread in the LAN setting, we obtain an online running time of $3ms$ and an overall time of 0.2 seconds. The next fastest is that of DH-based DKT protocol which required 1.7 seconds, an 8.5× slowdown compared to our protocol. For the larger set size of $n = 2^{12}$, our overall running time is 0.9 seconds with an online phase of just $40ms$. The DKT protocol is again the next fastest requiring 25× longer resulting in a total running time of 22.6 seconds. The DCW protocol from which ours is derived incurs more than a 60× overhead. For the largest set size performed of $n = 2^{20}$, our protocol achieves an online phase of 14 seconds and an overall time of 127 seconds. The DKT protocol overall running time was more than 95 minutes, a 47× overhead compared to our running time. The DCW protocol took prohibitively long to run but is expected to take more than 100× longer than our optimized protocol.

When evaluating our protocol in the WAN setting with $95ms$ round trip latency our protocol again achieves the fastest running times. For the small set size of $n = 2^8$, the protocol takes an overall running time of 0.95 seconds with the online phase taking 0.1 seconds. DKT was the next fastest protocol requiring a total time of 1.7 seconds, an almost 2× slowdown. Both variants of the DCW protocol experience a more significant slowdown of roughly 4×. When increasing the set size, our protocol experiences an even greater relative speedup. For $n = 2^{16}$, our protocol takes 56 seconds, with 11 of the seconds consisting of the online phase. Comparatively, DKT takes 393 seconds resulting in our protocol being more than 7× faster. The DCW protocols are even slower requiring more than 19 minutes, a 20× slowdown. This is primarily due to the need to perform the expensive secret-sharing operations and send more data.

In addition to faster serial performance, our protocol also benefits from easily being parallelized, unlike much of the DCW online phase. Figure Figure 3.3 shows the running times of our protocol and that of DKT when parallelized using p threads per party in the LAN setting. With $p = 4$ we obtain a speedup of 2.3× for set size $n = 2^{16}$ and 2× speedup for $n = 2^{20}$. However, the DKT protocol benefits from being trivially parallelizable. As such, they enjoy a nearly one-to-one speedup when more threads are used. This combined with the extremely small communication overhead of the DKT protocol could potentially allow their protocol to outperform ours when the network is quite slow and the parties have many threads available.

In Figure 3.4 we report the empirical and asymptotic communication costs of the protocols. Out of the bloom filter based protocols, ours consumes significantly less bandwidth. For $n = 2^8$, only 1.9MB communication was required with most of that cost in the offline phase. Then computing the intersection for $n = 2^{16}$, our protocol uses 324MB of communication, approximately 5KB per item. The largest amount of communication occurs during the OT extension and involves the sending of a roughly $2n\kappa^2$-bit matrix. The cut and choose contributes minimally to the communication and consists of np_{chk} choice bits and the xor of the corresponding OT messages. In the online phase, the sending of the permutation consisting of $N_{\mathsf{bf}} \log_2(N_{\mathsf{ot}}) \approx 2n\kappa \log(2n\kappa)$ bits that dominates the communication.

Parameters: X is Alice's input, Y is Bob's input. N_{bf} is the required Bloom filter size; k is the number of Bloom filter hash functions; N_{ot} is the number of OTs to generate. H is modeled as a random oracle with output length κ. The choice of these parameters, as well as others $\alpha, p_{\mathsf{chk}}, N_{\mathsf{maxones}}$, is described in Section 3.4.2.

1. **[setup]** The parties perform a secure coin-tossing subprotocol to choose (seeds for) random Bloom filter hash functions $h_1, \ldots, h_k : \{0,1\}^* \to [N_{\mathsf{bf}}]$.

2. **[random OTs]** Bob chooses a random string $b = b_1 \ldots b_{N_{\mathsf{ot}}}$ with an α fraction of 1s. Parties perform N_{ot} OTs of random messages (of length κ), with Alice choosing random strings $m_{i,0}, m_{i,1}$ in the ith instance. Bob uses choice bit b_i and learns $m_i^* = m_{i,b_i}$.

3. **[cut-and-choose challenge]** Alice chooses a set $C \subseteq [N_{\mathsf{ot}}]$ by choosing each index with independent probability p_{chk}. She sends C to Bob. Bob aborts if $|C| > N_{\mathsf{ot}} - N_{\mathsf{bf}}$.

4. **[cut-and-choose response]** Bob computes the set $R = \{i \in C \mid b_i = 0\}$ and sends R to Alice. To prove that he used choice bit 0 in the OTs indexed by R, Bob computes $r^* = \bigoplus_{i \in R} m_i^*$ and sends it to Alice. Alice aborts if $|C| - |R| > N_{\mathsf{maxones}}$ or if $r^* \neq \bigoplus_{i \in R} m_{i,0}$.

5. **[permute unopened OTs]** Bob generates a Bloom filter BF containing his items Y. He chooses a random injective function $\pi : [N_{\mathsf{bf}}] \to ([N_{\mathsf{ot}}] \setminus C)$ such that $BF[i] = b_{\pi(i)}$, and sends π to Alice.

6. **[randomized GBF]** For each item x in Alice's input set, she computes a summary value

$$K_x = H\left(x \,\Big\|\, \bigoplus_{i \in h_*(x)} m_{\pi(i),1} \right),$$

where $h_*(x) \overset{\text{def}}{=} \{h_i(x) : i \in [k]\}$. She sends a random permutation of $K = \{K_x \mid x \in X\}$.

7. **[output]** Bob outputs $\{y \in Y \mid H(y \,\|\, \bigoplus_{i \in h_*(y)} m^*_{\pi(i)}) \in K\}$.

Figure 3.2: Malicious-secure PSI protocol based on garbled Bloom filters.

Chapter 4

PSI From Dual Execution

Malicious-Secure Private Set Intersection via Dual Execution by Peter Rindal & Mike Rosulek, in CCS[RR17b].

4.1 Introduction

4.1.1 Chapter Contributions

From the previous discussion, we see that the fastest PSI paradigm for semi-honest security is due to Pinkas, Schneider, Zohnar[PSZ14] has no fully malicious-secure variant. We fill this gap by presenting a protocol based on the PSZ paradigm that achieves malicious-secure private set intersection.

We start with the observation that in the PSZ paradigm the two parties take the roles of Alice and Bob, and it is relatively straight-forward to secure the protocol against a malicious Bob [OOS17]. Therefore our approach is to run the protocol in both directions, so that each party must play the role of receiver at different times in the protocol. This high-level idea is inspired by the *dual-execution* technique of Mohassel & Franklin [MF06a]. In that work, the parties perform two executions of Yao's protocol in opposite directions, taking advantage of the fact that Yao's protocol is easily secured against a malicious Bob. In that setting, the resulting dual-execution protocol achieves malicious security but leaks one adversarially-chosen bit. In our setting, however, we are able to carefully combine the two PSI executions in a way that achieves the usual notion of *full* malicious security.

Because our protocol is based on the fast PSZ paradigm, it relies exclusively on cheap symmetric-key cryptography. We have implemented our protocol and compare it to the previous state of the art. We find our protocol to be 12× faster than the previous fastest malicious-secure PSI protocol of [RR17a], on large datasets. Our implementation can securely compute the intersection of million-item sets in only 12.6 seconds on a single thread (2.9 seconds with many threads).

Finally, as mentioned above, the previous fastest malicious PSI protocol [RR17a] appears to rely inherently on the random-oracle model. We show that our protocol can be instantiated in the standard model. Both our standard model and random-oracle optimized protocols are faster than [RR17a] in the LAN setting, with our latter protocol being the fastest across all settings.

4.2 Overview of PSZ Paradigm

Pinkas, Schneider, and Zohner [PSZ14] (hereafter PSZ) introduced a paradigm for PSI that is secure against semi-honest adversaries. There have since been several improvements made to this general paradigm [PSSZ15, KKRT16, OOS17]. In particular, the implementation of [KKRT16] is the fastest secure PSI protocol to date. Adapting this paradigm to the malicious security model is therefore a natural direction.

In this section, we describe the PSZ paradigm, and discuss what prevents it from achieving malicious security.

4.2.1 High-Level Overview

The PSZ paradigm works as follows. First, for simplicity suppose Alice has n items $X = \{x_1, \ldots, x_n\}$ while Bob has one item y. The goal of *private set inclusion* is for Bob to learn whether $y \in X$, and nothing more. We abstract the main step of the protocol as an **oblivious encoding** step, which is similar in spirit to an oblivious pseudorandom function [FIPR05]. The parties interact so that Alice learns a random mapping F, while Bob learns only $F(y)$. The details of this step are not relevant at the moment. Then Alice sends $F(X) = \{F(x_1), \ldots, F(x_n)\}$ to Bob. Bob can check whether his value $F(y)$ is among these values and therefore learn whether $y \in \{x_1, \ldots, x_n\}$.

Since F is a random mapping, the other items in $F(X)$ leak nothing about the set X.

The protocol can be extended to a proper PSI protocol, where Bob has a set of items $Y = \{y_1, \ldots, y_n\}$. The parties simply perform n instances of the private set inclusion protocol, one for each y_i, with Alice using the same input X each time. This leads to a PSI protocol with $O(n^2)$ communication.

To reduce the communication cost, the parties can agree on a hashing scheme that assigns their items to bins. In PSZ, they propose to use a variant of Cuckoo hashing. For the sake of example, suppose Bob uses cuckoo hashing with two hash functions to assign his items to bins. In cuckoo hashing, Bob will assign item y to the bin with index either $h_1(y)$ or $h_2(y)$, so that each bin contains at most one item. Alice will assign each of her items x to *both* bins $h_1(x)$ and $h_2(x)$, so that each of her bins may contain several items. Overall, for each bin Alice has several items while Bob has (at most) one, so they can perform the private set inclusion protocol for each bin. There are of course many details to work out, but by using this main idea the communication cost of protocol can be reduced to $O(n)$.

4.2.2 Insecurity against Malicious Adversaries

The PSZ protocol and its followups are proven secure in the semi-honest setting, but are not secure against malicious adversaries. There are several features of the protocol that present challenges in the presence of malicious adversaries:

- Even if the "oblivious encoding" subprotocol is made secure against malicious adversaries, the set-inclusion subprotocol does not become malicious-secure. The technical challenge relates to the problem of the simulator extracting inputs from a malicious Alice. The simulator sees only the random mapping F and the items $\{F(x_1), \ldots, F(x_n)\}$ sent by Alice. For the simulator to extract Alice's effective input, the mapping F must be invertible. However, the oblivious encoding instantiations generally do not result in an invertible F.
- In the PSZ protocol, Bob uses cuckoo hashing to assign his items to bins. Each item y may be placed in two possible locations, and the final placement of item *y depends on all of Bob's other items.* A corrupt Alice may exploit this in the protocol to learn information about Bob's

set. In particular, Alice is supposed to place each item x in *both* possible locations $h_1(x)$ and $h_2(x)$. A corrupt Alice may place x only in $h_1(x)$. Then if x turns out to be in the intersection, Alice learns that Bob placed x in $h_1(x)$ but not $h_2(x)$. As just mentioned, whether Bob places an item according to h_1 or h_2 depends on all of Bob's items, so it is information that cannot be simulated in the ideal world.

- In the $O(n^2)$ PSI protocol, Alice is supposed to run many instances of the simple set-inclusion protocol with the same set X each time. However, a malicious Alice may use different sets in different instances. In doing so, she can influence the output of the protocol in ways that cannot be simulated in the ideal world.

4.3 Oblivious Encoding

As discussed in the previous section, the PSZ paradigm uses an **oblivious encoding** step. In Figure 4.1 we define an ideal functionality for this task. Intuitively, the functionality chooses a random mapping F, allows the receiver to learn $F[c]$ for a single c, and allows the sender to learn $F[c]$ for an unlimited number of c's. However, if the sender is corrupt, the functionality allows the sender to *choose* the mapping F (so that it need not be random). This reflects what our instantiations of this functionality are able to achieve.

We describe two instantiations of this functionality that are secure in the presence of malicious adversaries.

In the programmable-random-oracle model. Orrù, Orsini & Scholl [OOS17] describe an efficient 1-out-of-N oblivious transfer protocol, for random OT secrets and N exponential in the security parameter. The protocol is secure against malicious adversaries. In order to model an exponential number of OT secrets, they give an ideal functionality which is identical to ours except that the adversary is never allowed to choose the mapping. Hence, their protocol also realizes our functionality as well (the simulator simply chooses $F_{\mathsf{adv}} = \perp$ so that the functionality always chooses a random mapping).

Their protocol is proven secure in the programmable random-oracle model. Concretely, the cost of a single OT/oblivious encoding in their protocol is roughly 3 times that of a single semi-honest 1-out-of-2 OT.

Parameters: two parties denoted as Sender and Receiver. The input domain $\{0,1\}^\sigma$ and output domain $\{0,1\}^\ell$ for a private F.

1. [**Initialization**] Create an initially empty associative array $F : \{0,1\}^\sigma \to \{0,1\}^\ell$.
2. [**Receiver Encode**] Wait for a command $(\textsc{Encode}, \mathsf{sid}, c)$ from the Receiver, and record c. Then:
3. [**Adversarial Map Choice**] If the sender is corrupt, then send $(\textsc{RecvInput}, \mathsf{sid})$ to the adversary and wait for a response of the form $(\textsc{Deliver}, \mathsf{sid}, F_{\mathsf{adv}})$. If the sender is honest, set $F_{\mathsf{adv}} = \bot$. Then:
4. [**Receiver Output**] If $F_{\mathsf{adv}} = \bot$ then choose $F[c]$ uniformly at random; otherwise set $F[c] := F_{\mathsf{adv}}(c)$, interpreting F_{adv} as a circuit. Give $(\textsc{Output}, \mathsf{sid}, F[c])$ to the receiver. Then:
5. [**Sender Encode**] Stop responding to any requests by the receiver. But for any number of commands $(\textsc{Encode}, \mathsf{sid}, c')$ from the sender, do the following:
   - If $F[c']$ doesn't exist and $F_{\mathsf{adv}} = \bot$, choose $F[c']$ uniformly at random.
   - If $F[c']$ doesn't exist and $F_{\mathsf{adv}} \neq \bot$, set $F[c'] := F_{\mathsf{adv}}(c')$.
   - Give $(\textsc{Output}, \mathsf{sid}, c', F[c'])$ to the sender.

Figure 4.1: The Oblivious Encoding ideal functionality $\mathcal{F}_{\mathsf{encode}}$

In the standard model. In the standard model, it is possible to use a variant of the semi-honest oblivious encoding subprotocol from PSZ. The protocol works as follows, where the receiver has input c:

- The sender chooses 2σ random κ-bit strings: $m[1,0], m[1,1], \ldots, m[\sigma,0], m[\sigma,1]$.
- The parties perform σ instances of OT, where in the ith instance the sender provides inputs $m[i,0], m[i,1]$, the receiver provides input c_i and receives $m[i,c_i]$.
- The receiver computes output $\bigoplus_i \mathsf{PRF}(m[i,c_i],c)$, where PRF is a secure pseudorandom function with ℓ bits of output.

- To obtain the encoding of any value c', the receiver can compute $\bigoplus_i \mathsf{PRF}(m[i, c'_i], c')$.

For security against a corrupt receiver, the simulator can extract c from the receiver's OT inputs. We can then argue that all other oblivious encodings look random to the receiver. Indeed, for every $c' \neq c$, there is a position i in which $c'_i \neq c_i$, so the corresponding encoding $\bigoplus_i \mathsf{PRF}(m[i, c'_i], c')$ contains a term $\mathsf{PRF}(m[i, c'_i], c')$ that is random from the receiver's point of view.

For security against a corrupt sender, the simulator can extract the $m[i, b]$ values from the sender's OT inputs. It can then hard-code these values into a circuit $F_{\mathsf{adv}}(c) = \bigoplus_i \mathsf{PRF}(m[i, c_i], c)$ and send this circuit to the ideal functionality.

The cost of this protocol is σ instances of OT per oblivious encoding. Since the protocol uses OTs with *chosen* secrets (not random secrets chosen by the functionality), it can be instantiated in the standard model.[1]

4.4 A Warmup: Quadratic-Cost PSI

The main technical idea for achieving malicious security is to carefully apply the dual execution paradigm of Mohassel & Franklin [MF06a] to the PSZ paradigm for private set intersection. In this section we give a protocol which contains the main ideas of our approach, but which has quadratic complexity. In the next section we describe how to apply a hashing technique to reduce the cost.

4.4.1 Dual Execution Protocol

The main idea behind our approach is as follows (a formal description is given in Figure 4.2):

1. The parties perform an encoding step similar to PSZ, where Alice acts as receiver. In more detail, the parties invoke $\mathcal{F}_{\mathsf{encode}}$ once for each of

[1] Modern OT extension protocols can be optimized for OT of random secrets, but it is not known how to make this special case less expensive while avoiding the programmable-random-oracle model.

Alice's items. Alice learns $[\![x_j]\!]_j^{\mathsf{B}}$, where x_j is her jth item and $[\![\cdot]\!]_j^{\mathsf{B}}$ is the encoding used in the jth instance of $\mathcal{F}_{\mathsf{encode}}$. Note that Bob can obtain $[\![v]\!]_j^{\mathsf{B}}$ for any v and any j, by appropriately querying the functionality.

2. The parties do the same thing with the roles reversed. Bob learns $[\![y_i]\!]_i^{\mathsf{A}}$, where y_i is his ith item and $[\![\cdot]\!]_i^{\mathsf{A}}$ is the encoding. As above, Alice can obtain any encoding of the form $[\![v]\!]_j^{\mathsf{A}}$.

At this point, let us define a **common encoding**:

$$[\![v]\!]_{i,j} \stackrel{\text{def}}{=} [\![v]\!]_i^{\mathsf{A}} \oplus [\![v]\!]_j^{\mathsf{B}}$$

The important property of this encoding is:

- If Alice knows $[\![v]\!]_j^{\mathsf{B}}$ then she can compute the common encoding $[\![v]\!]_{i,j}$ for any i.
- If Alice does not know $[\![v]\!]_j^{\mathsf{B}}$, then it is actually random from her point of view. It is therefore hard for her to predict common encoding $[\![v]\!]_{i,j}$ for any i.

A symmetric condition holds for Bob. Now the idea is for the parties to compute all of the common encodings that they can deduce from these rules. Then the intersection of these encodings will correspond to the intersection of their sets. In other words (continuing the protocol overview):

3. Alice computes a set of encodings $E = \{[\![x_j]\!]_{i,j} \mid i, j \in [n]\}$, and sends it to Bob.
4. Bob likewise computes a set of encodings and checks which of them appear in E. These encodings correspond to the intersection. More formally, Bob outputs:

$$Z = \{y_i \in Y \mid \exists j \in [n] : [\![y_i]\!]_{i,j} \in E\}$$

We note that in this protocol, only Bob receives output. In fact, it turns out to be problematic if Bob sends an analogous set of encodings to Alice. In Section 4.5.7 we discuss in more detail the problems associated with both parties receiving output.

4.4.2 Security

The protocol achieves malicious security:

Theorem 3. *The protocol in Figure 4.2 is a UC-secure protocol for PSI in the $\mathcal{F}_{\mathsf{encode}}$-hybrid model.*

We defer giving a formal proof for this protocol in favor of a single proof of our final protocol in the next section. Instead, we sketch the high-level idea of the simulation.

When Alice is corrupt, the simulator plays the role of $\mathcal{F}_{\mathsf{encode}}$ and therefore observes Alice's inputs to the functionality during Step 2. Let x_j denote Alice's jth input to $\mathcal{F}_{\mathsf{encode}}$, in which she learns $[\![x_j]\!]^{\mathsf{B}}_j$. Let $\tilde{X} = \{x_1, \ldots, x_n\}$. We can make the following observations:

- Suppose Bob has an item $y \notin \tilde{X}$. In the protocol, Alice will send a set of encodings E, and Bob will search this set for encodings $[\![y]\!]_{i,j}$, for certain i, j values. But by the definition of $\tilde{X}$, Alice does not know any encoding of the form $[\![y]\!]^{\mathsf{B}}_j$, and so with high probability cannot guess any encoding which will cause Bob to include y in the output. In other words, we can argue that **Alice's effective input is a subset of $\tilde{X}$.**

- Suppose for simplicity Bob's input happens to be $\tilde{X}$. This turns out to be the most interesting case for the proof. Bob will randomly permute these items and obtain an encoding of each one. Let π be the permutation such that Bob learns $[\![x_j]\!]^{\mathsf{A}}_{\pi(j)}$. Now Bob will be looking in the set E for common encodings of the form $[\![x_j]\!]_{\pi(j),*}$. Note that from the definition of $\tilde{X}$, Alice can only produce valid encodings of the form $[\![x_j]\!]_{*,j}$. It follows that **Bob will include a value x_j in his output if and only if Alice includes encoding $[\![x_j]\!]_{\pi(j),j} \in E$.**

Since the distribution of π is random, the simulator can simulate the effect. More precisely, the simulator chooses a random π and sets $X^* = \{x_j \mid [\![x_j]\!]_{\pi(j),j} \in E\}$. It is this X^* that the simulator finally sends to the ideal functionality. In the above, we were considering a special case where Bob's input happens to be $\tilde{X}$. However, this simulation approach works in general.

The simulation for a malicious Bob is simpler, and it relies on the fact that common encodings look random, for values not in the intersection.

Parameters: $\mathcal{F}_{\mathsf{encode}}$ is the Oblivious Encoding functionality with input domain $\{0,1\}^\sigma$ output bit length $\lambda + 2\log n$.

On Input $(\textsc{Send}, \mathsf{sid}, X)$ from Alice and $(\textsc{Receive}, \mathsf{sid}, Y)$ from Bob, where $X, Y \subseteq \{0,1\}^\sigma$ and $|X| = |Y| = n$. Each party randomly permutes their set.

1. [**A Encoding**] For $i \in [n]$, Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, i), y_i)$ to $\mathcal{F}_{\mathsf{encode}}$ who sends $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, i), \llbracket y_i \rrbracket_i^{\mathsf{A}})$ to Bob and $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, i))$ to Alice.

   For $j \in [n]$, Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, i),\ x_j)$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, i),\ \llbracket x_j \rrbracket_i^{\mathsf{A}})$ in response.

2. [**B Encoding**] For $i \in [n]$, Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, i), x_i)$ to $\mathcal{F}_{\mathsf{encode}}$ who sends $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, i), \llbracket x_i \rrbracket_i^{\mathsf{B}})$ to Alice and $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, i))$ to Bob.

   For $j \in [n]$, Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, i),\ y_j)$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, i),\ \llbracket y_j \rrbracket_i^{\mathsf{B}})$ in response.

3. [**Output**] Alice sends the common encodings

   $$E = \{\llbracket x_j \rrbracket_i^{\mathsf{A}} \oplus \llbracket x_j \rrbracket_j^{\mathsf{B}} \mid i, j \in [n]\}$$

   to Bob who outputs

   $$\{y_i \mid \exists j : \llbracket y_i \rrbracket_i^{\mathsf{A}} \oplus \llbracket y_i \rrbracket_j^{\mathsf{B}} \in E\}$$

Figure 4.2: Malicious-secure n^2 PSI protocol.

The protocol is correct as long as there are no spurious collisions among common encodings. That is, we do not have any $x_j \in X$ and $y_i \in Y \setminus X$ for which $\llbracket x_j \rrbracket_{i,j} = \llbracket y_i \rrbracket_{i,j}$ (which would cause Bob to erroneously place y_i in the intersection). The probability of this happening for a fixed x_j, y_i is $2^{-\ell}$, if the encodings have length ℓ. By a union bound, the total probability of such an event is $n^2 2^{-\ell}$. We set $\ell = \lambda + 2\log n$ to ensure this error probability is at most $2^{-\lambda}$.

4.4.3 Encode-Commit Protocol

In addition to the approach described above, we present an alternative protocol based on $\mathcal{F}_{\mathsf{encode}}$ and commitments that offers communication/computation trade-offs. Fundamentally, the dual execution protocol above first restricts Alice to her set by requiring her to encode it as $\{[\![x_1]\!]^{\mathsf{B}}_1, ..., [\![x_n]\!]^{\mathsf{B}}_n\}$. In some sense this encoding operation can be viewed as Alice committing to her inputs. The property that we need from the B encoding are: 1) $[\![*]\!]^{\mathsf{B}}$ must allow the simulator to extract the set of candidate x_j values; 2) provides a binding proof to the value x_j. Continuing to view $[\![*]\!]^{\mathsf{B}}$ as a commitment, the dual execution protocol instructs Alice to then decommit (prove she was bound to x_j) to these values by sending all $[\![x_j]\!]^{\mathsf{B}}_j$ encodings to Bob, but masked under $[\![x_j]\!]^{\mathsf{A}}_i$ so that the commitment can only be "decommitted" if Bob knows *one* of these encodings of x_j.

Taking this idea to its conclusion, we can formulate a new protocol where Alice simply commits to her inputs by sending $\textsc{Comm}(x_1; r_1), ..., \textsc{Comm}(x_n; r_n)$ to Bob in lieu of Figure 4.2 Step 2, where $\textsc{Comm}$ is a standard (non-interactive) commitment scheme. The final step of the protocol is for her to send the decommitment r_j masked under the encodings of x_j

$$E = \left\{[\![x_j]\!]^{\mathsf{A}}_i \oplus r_j \mid i, j \in [n]\right\}$$

In the event that Bob knows $[\![x_j]\!]^{\mathsf{A}}_i$, i.e. his input contains $y_i = x_j$, he will be able to recover the decommitment value r_j and decommit $\textsc{Comm}(x_j; r_j)$, thereby inferring that x_j is in the intersection.

The security proof of this protocol follows the same structure as before. For the more interesting case of a malicious Alice, we require an extractable commitment scheme. The simulator is able to extract the set $\tilde{X} = \{x_1, ..., x_n\}$ from the commitments $\textsc{Comm}(x_1; r_1), ..., \textsc{Comm}(x_n; r_n)$ and sends $X^* = \{x_j \mid [\![x_j]\!]^{\mathsf{A}}_{\pi(j)} \oplus r_j \in E\}$ to the functionality. The correctness of this simulation strategy follows from the sketch in the previous section by viewing $\textsc{Comm}(x_j; r_j)$ as equivalent to the encoding $[\![*]\!]^{\mathsf{B}}_j$ and r_j as equivalent to $[\![x_j]\!]^{\mathsf{B}}_j$.

The communication and computation complexity for both of these protocols is $O(n^2)$. However, we will later show that the concrete communication/computation overheads of these two approaches result in interesting performance trade-offs. Most notable is that the commitment based approach requires less computation at the expense of additional communication, making it more efficient in the LAN setting.

4.5 Our Full Protocol

After constructing a quadratic-cost PSI protocol, the PSZ paradigm is for the parties to use a hashing scheme to assign their items into *bins*, and then perform the quadratic-cost PSI on each bin. We review this approach here, and discuss challenges specific to the malicious setting.

4.5.1 Hashing

Cuckoo hashing and its drawbacks. The most efficient hashing scheme in PSZ is Cuckoo hashing. In this approach, the parties agree on two (or more) random functions h_1 and h_2. Alice uses Cuckoo hashing to map her items into bins. As a result, each item x is placed in either bin $\mathcal{B}[h_1(x)]$ or $\mathcal{B}[h_2(x)]$ such that each bin has at most one item. Bob conceptually places each of his items y into *both* bins $\mathcal{B}[h_1(y)]$ or $\mathcal{B}[h_2(y)]$. Then the parties perform a PSI for the items in each bin. Since Alice has only one item per bin, these PSIs are quite efficient.

Unfortunately, this general hashing approach does not immediately work in the malicious security setting. Roughly speaking, the problem is that Bob may place an item y into bin $\mathcal{B}[h_1(y)]$ but *not* in $\mathcal{B}[h_2(y)]$. Suppose Alice also has item y, then y will appear in the output if and only if Alice's cuckoo hashing has chosen to place it in $\mathcal{B}[h_1(y)]$ and not $\mathcal{B}[h_2(y)]$. Because of the nature of Cuckoo hashing, whether an item is placed according to h_1 or h_2 event depends in a subtle way on *all other items* in Alice's set. As a result, the effect of Bob's misbehavior cannot be simulated in the ideal world.

Simple hashing. While Cuckoo hashing is problematic for malicious security, we can still use a *simple hashing* approach. The parties agree on a random function $h : \{0,1\}^* \to [m]$ and assign item x to bin $\mathcal{B}[h(x)]$. Then parties can perform a PSI for each bin. Note that under this hashing scheme, the hashed location of each item does not depend on other items in the set. Each item has only one "correct" location.

Note that the load (number of items assigned) of any bin leaks some information about a party's input set. Therefore, all bins must be padded to some maximum possible size. A standard balls-and-bins argument shows that the maximum load among the $m = O(n/\log n)$ bins is $O(\log n)$ with very high probability.

Phasing. In the standard-model variant of our protocol, the oblivious encoding step scales linearly with the length of the items being encoded. Our random-oracle protocol also has a weak dependence on the representation length of the items which is affected by the size of the sets. Hence, it is desirable to reduce the length of these items as much as possible.

Pinkas et al. [PSSZ15] described how to use a hashing technique of Arbitman et al. [ANS10a] called **phasing** (permutation-based hashing) to reduce the length of items in each bin. The idea is as follows. Suppose we are considering PSI on strings of length σ bits. Let h be a random function with output range $\{0,1\}^d$, where the number of bins is 2^d. To assign an item x to a bin, we write $x = x_L \| x_R$, with $|x_L| = d$. We assign this item to bin $h(x_R) \oplus x_L$, and store it in that bin with x_R as its representation. Arbitman et al. [ANS10a] show that this method of assigning items to bins results in maximum load $O(\log n)$ with high probability.

Note that the representations in each bin are $\sigma - d$ bits long — shorter by d bits. Importantly, shrinking these representations does not introduce any collisions. This is because the mapping $\mathsf{phase}(x_L \| x_R) = (h(x_R) \oplus x_L, x_R)$ is a Feistel function and therefore invertible. So distinct items will either be mapped to distinct bins, or, in the case that they are mapped to the same bin, they *must* be assigned different representations. Hence the PSI subprotocol in each bin can be performed on the shorter representations.

The idea can be extended as follows, when the number m of bins is not a power of two (here h is taken to be a function with range $[m]$):

$$\begin{aligned} \mathsf{phase}_{h,m}(x) &= \Big(h(\lfloor x/m \rfloor) + x \bmod m, \ \ \lfloor x/m \rfloor \Big) \\ \mathsf{phase}^{-1}_{h,m}(b,z) &= zm + [h(z) + b \bmod m] \end{aligned}$$

We show that phasing is a secure way to reduce the length of items, in the presence of malicious adversaries.

4.5.2 Aggregating Masks Across Bins

Suppose we apply the simple hashing technique to our quadratic PSI protocol. The resulting protocol would work as follows.

1. First, the parties hash their n items into $m = O(n/\log n)$ bins. With high probability each bin has at most $\mu = O(\log n)$ items. Bins are

artificially padded with dummy items to a fixed size of μ items.

2. For each bin the parties perform the quadratic-cost PSI protocol from Section 4.4. Each party acts as $\mathcal{F}_{\mathsf{encode}}$ sender and receiver, and computes common encodings of the items. For each bin, Alice sends all $\mu^2 = O(\log^2 n)$ encodings to Bob, who computes the intersection.

The total cost of this protocol is therefore $m\mu^2 = O(n \log n)$, a significant improvement over the quadratic protocol.

We present an additional optimization which reduces the cost by a significant constant factor. Our primary observation is that in order to hide the number of items in each bin, the parties must pad the bins out to the maximum size μ. However, this results in their bins containing mostly dummy items (in our range of parameters, around 75% are dummy items).

When Alice sends her common encodings in the final step of the protocol, she knows that the encodings for dummy items cannot contribute to the final result. If she had a way to avoid sending these dummy encodings, it would reduce the number of encodings sent by roughly a factor of 4.

Hence, we suggest an optimization in which Alice aggregates her encodings *across all the bins*, and send only the *non-dummy* encodings to Bob, as a unified collection. Similarly, Bob need not check Alice's set of encodings for one of his dummy encodings. So Bob computes common encodings only for his actual input items.

To show the security of this change, we need only consider Bob's view which has been slightly altered. Suppose Alice chooses a random value d to be a "universal" dummy item in each bin. Since this item is chosen randomly, it is negligibly likely that Bob would have used it as input to any instance of $\mathcal{F}_{\mathsf{encode}}$ where he was the receiver. Hence, the common encodings of dummy values look random from Bob's perspective. Intuitively, the only common encodings we removed from the protocol are ones that looked random from Bob's perspective (and hence, had no effect on his output, with overwhelming probability).

Note that it is not secure to eliminate dummy encodings within a *single* quadratic-PSI. This would leak how many items Alice assigned to that bin. It is not secure to leak the number of items in each bin. (It is for this reason that we still must perform exactly μ oblivious encoding steps per bin.) However, it is safe to leak the fact that Alice has n items *total.* By aggregating encodings across *all* bins we are able to use this common knowledge. Bob now sees a single collection of $n\mu$ encodings, but does not know which bins they correspond to.

After making this change, Bob is comparing each of his $n\mu$ non-dummy encodings to each of Alice's $n\mu$ encodings. Without this optimization, he only compares encodings within each bin. With more comparisons made among the common encodings, the probability of spurious collisions increases. We must therefore increase the length of these encodings. A similar argument to the previous section shows that if the encodings have length $\lambda + 2\log(n\mu)$, then the overall probability of a spurious collision is $2^{-\lambda}$.

4.5.3 Dual Execution Protocol Details & Security

The formal details of our dual execution protocol are given in Figure 4.3. The protocol follows the high-level outline developed in this section. We use the following notation:

- $[\![x]\!]^{\mathsf{A}}_{b,p}$ denotes an encoding of value x, in an instance of $\mathcal{F}_{\mathsf{encode}}$ where Alice is sender, corresponding to position p in bin b. Each bin stores a maximum of μ items, so there are μ positions.
- We write $(b, x') = \mathsf{phase}_{h,m}(x)$ to denote the phasing operation (Section 4.5.1), where to store item x we place representative x' in bin b.

Theorem 4. *The protocol in Figure 4.3 is UC-secure in the $\mathcal{F}_{\mathsf{encode}}$-hybrid model. The resulting protocol has cost $O(Cn\log n)$, where $C \approx \kappa$ is the cost of one $\mathcal{F}_{\mathsf{encode}}$ call on a $\sigma - \log n$ length bit string.*

Proof. We start with the case of a **corrupt Bob**. The simulator must extract Bob's input, and simulate the messages in the protocol. We first describe the simulator:

The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality. The simulator does nothing in Step 2 and Step 3a (steps where Bob receives no output). To extract Bob's set, the simulator observes all of Bob's $\mathcal{F}_{\mathsf{encode}}$ messages $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, p), y'_{b,p})$ in Step 3b. The simulator computes $Y = \{\mathsf{phase}^{-1}_{h,m}(b, y'_{b,p}) \mid b \in [m], p \in [\mu]\}$ and sends it to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality which responds with the intersection $Z = X \cap Y$.

Set Z^* to be equal to Z along with arbitrary dummy items not in Y, so that $|Z^*| = n$. For each $z \in Z^*$, compute $(b, z') = \mathsf{phase}_{m,h}(z)$ and insert z' into a random unused position bin $\mathcal{B}_X[b]$. For $z \in Z^*$ in random order, and $j \in [\mu]$, compute $(b, z') = \mathsf{phase}_{m,h}(z)$ and send $[\![z']\!]^{\mathsf{A}}_{b,p} \oplus [\![z']\!]^{\mathsf{B}}_{b,j}$ to Bob, where these encodings are obtained by playing the role of $\mathcal{F}_{\mathsf{encode}}$.

To show that this is a valid simulation, we consider a series of hybrids.

Hybrid 0 The first hybrid is the real interaction as specified in Figure 4.2 where Alice honestly uses her input X, and $\mathcal{F}_{\mathsf{encode}}$ is implemented honestly.

Observe Bob's commands to $\mathcal{F}_{\mathsf{encode}}$ of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, p), y'_{b,p})$ in Step 3b. Based on these, define the set $\tilde{Y} = \{\mathsf{phase}^{-1}_{h,m}(b, y'_{b,p}) \mid b \in [m], p \in [\mu]\}$.

Hybrid 1 In this hybrid, we modify Alice to send dummy values to $\mathcal{F}_{\mathsf{encode}}$ in Step 2a. Then we further modify Alice to perform the hashing at the last possible moment in Step 4. The simulation can obtain the appropriate encodings directly from the simulated $\mathcal{F}_{\mathsf{encode}}$. The hybrid is indistinguishable by the properties of $\mathcal{F}_{\mathsf{encode}}$.

Hybrid 2 In Step 4a, for each $x \in X$ the simulated Alice sends common encodings of the form $[\![x']\!]^{\mathsf{A}}_{b,j} \oplus [\![x']\!]^{\mathsf{B}}_{b,p}$, for some position p, where $(b, x') = \mathsf{phase}_{h,m}(x)$. Suppose $x \notin \tilde{Y}$. By construction of $\tilde{Y}$, Bob never obtained an encoding of the form $[\![x']\!]^{\mathsf{A}}_{b,j}$. This encoding is therefore distributed independent of everything else in the simulation. In particular, the *common* encodings corresponding to this x are distributed independently of the choice of (b, x') and hence the choice of x.

We therefore modify the hybrid in the following way. Before Alice adds the items of X to her hash table in Step 4a, she replaces all items in $X \setminus \tilde{Y}$ (*i.e.*, all items not in $X \cap \tilde{Y}$) with fixed dummy values not in Y. By the above argument, the adversary's view is identically distributed in this modified hybrid.

The final hybrid works as follows. A simulator interacts with the adversary and determines a set $\tilde{Y}$, without using Alice's actual input X. Then it computes $X \cap \tilde{Y}$ and simulates Alice's message in Step 4a using only $X \cap \tilde{Y}$. Hence, this hybrid corresponds to our final simulator, where we send $\tilde{Y}$ to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality and receive $X \cap \tilde{Y}$ in response.

We now turn our attention to a **corrupt Alice**. In this case the simulator must simply extract Alice's effective input (Alice receives no output from $\mathcal{F}_{\mathsf{PSI}}$). The simulator is defined as follows:

The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality. The simulator does nothing in Step 2 and Step 3b. In Step 3a, the simulator intercepts Alice's commands of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, b, p), x'_{b,p})$. The simulator computes a set of candidates $\tilde{X} = \{\mathsf{phase}^{-1}_{h,m}(b, x'_{b,p}) \mid b \in [m], p \in [\mu]\}$ and for $x \in \tilde{X}$ let $\mathsf{c}(x)$ denote the number of times that $\mathsf{phase}^{-1}_{h,m}(b, x'_{b,p}) = x$ for $b \in [m], p \in [\mu]$.

The simulator computes a hash table $\mathcal{B}$ as follows. For $x \in \tilde{X}$ and $i \in \mathsf{c}(x)$, the simulator computes $(b, x') = \mathsf{phase}_{h,m}(x)$ and places x' in a random unused position in bin $\mathcal{B}[b]$. Although $|\tilde{X}|$ may be as large as $m\mu$, by construction no bin will have more than μ items. For each such x, let $\mathsf{p}(x)$ denote the set of positions of x in its bin.

Let E denote the set of values sent by Alice in Step 4a. The simulator computes

$$\begin{aligned} X^* = \big\{x \in \tilde{X} \mid \exists j \in [\mu], p \in \mathsf{p}(x) :& \\ [\![x']\!]^{\mathsf{A}}_{b,j} \oplus [\![x']\!]^{\mathsf{B}}_{b,p} \in E& \\ \wedge\, (b, x') = \mathsf{phase}_{h,m}(x)\big\}& \end{aligned} \tag{4.1}$$

where the encodings are obtained by playing the role of $\mathcal{F}_{\mathsf{encode}}$. The simulator sends X^* to the $\mathcal{F}_{\mathsf{PSI}}$ functionality.

Hybrid 0 The first hybrid is the real interaction as specified in Figure 4.2 where Bob honestly uses his input X, and $\mathcal{F}_{\mathsf{encode}}$ is implemented honestly.

Observe Alice's commands to $\mathcal{F}_{\mathsf{encode}}$ of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, b, p), x'_{b,p})$ in Step 3a. Based on these, define $\tilde{X} = \{\mathsf{phase}^{-1}_{h,m}(b, x'_{b,p}) \mid b \in [m], p \in [\mu]\}$.

Hybrid 1 In this hybrid, we modify Bob to send the zero string to $\mathcal{F}_{\mathsf{encode}}$ in Step 2b. The simulation can obtain all required encodings directly from the simulated $\mathcal{F}_{\mathsf{encode}}$. We also have Bob perform his hashing not in Step 2b but at the last possible moment in Step 4b. The hybrid is indistinguishable by the properties of $\mathcal{F}_{\mathsf{encode}}$.

Hybrid 2 The hybrid computes the output as specified in Step 4b. We then modify it to immediately remove all from this output which is not in $\tilde{X}$. The hybrids differ only in the event that simulated Bob computes an output in Step 4b that includes an item $y \notin \tilde{X}$. This happens only if $[\![y']\!]^{\mathsf{A}}_{b,j} \oplus [\![y']\!]^{\mathsf{B}}_{b,p} \in E$, where $(b, y') = \mathsf{phase}_{h,m}(y)$ and Bob places y' in position p. Since $y \notin \tilde{X}$, however, the encoding $[\![y']\!]^{\mathsf{B}}_{b,p}$ is distributed uniformly. The length of encodings is chosen so that the overall probability of this event (across all choices of $y \notin \tilde{X}$) is at most $2^{-\lambda}$. Hence the modification is indistinguishable.

Hybrid 3 We modify the hybrid in the following way. When building the hash table in Step 4b, the simulated Bob uses $\tilde{X}$ instead of his actual input Y. Each $x \in \tilde{X}$ is inserted $\mathsf{c}(x)$ times. Then he computes the protocol output as specified in Step 4b; call it X^*. This is not what the simulator gives as output — rather, it gives $X^* \cap Y$ as output instead.

The hashing process is different only in the fact that items of $Y \setminus \tilde{X}$ are excluded and replaced in the hash table with items of $\tilde{X} \setminus Y$ (*i.e.*, items in $Y \cap \tilde{X}$ are treated exactly the same way). Note that the definition of $\tilde{X}$ ensures that the hash table can hold all of these items without overflowing. Also, this change is local to Step 4b, where the only thing that happens is Bob computing his output. However, by the restriction added in *Hybrid 2* , items in $Y \setminus \tilde{X}$ can never be included in X^*. Similarly, by the step added in this hybrid, items in $\tilde{X} \setminus Y$ can never be included in the simulator's output. So this change has no effect on the adversary's view (which includes this final output).

The final hybrid works as follows. A simulator interacts with the adversary and at some point computes a set X^*, without the use of Y. Then the simulated Bob's output is computed as $X^* \cap Y$. Hence, this hybrid corresponds to our final simulator, where we send X^* to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality, which sends output $X^* \cap Y$ to ideal Bob. □

Set Size for Malicious Parties As the ideal PSI functionality in Figure 2.1 indicates, our protocol realized a slightly relaxed variant of traditional PSI that does not strictly enforce the size of a corrupt party's input set. The functionality allows an honest party to provide an input set of size n, but a corrupt party to provide a set of size $n' > n$. We now analyze why this is the case and what is the exact relationship between n and n'.

Let us first consider the case of a malicious Bob who learns the intersection. The simulator extracts a set based on the commands Bob gave when acting as $\mathcal{F}_{\mathsf{encode}}$ receiver. Bob is given $m\mu = O(n)$ opportunities to act as $\mathcal{F}_{\mathsf{encode}}$ receiver, and therefore the simulator extracts a set of size at most $n' = m\mu = O(n)$. Concretely, when $\lambda = 40, n = 2^{20}$ and $m = n/\log_2 n$, the optimal bin size is $\mu = 68$ and Bob's maximum set size is $n' < 4n$.

The situation for a malicious Alice is similar. As above, the simulator computes a set $\tilde{X}$ based on commands Alice gives to $\mathcal{F}_{\mathsf{encode}}$ when acting as receiver. The size of $\tilde{X}$ is therefore at most $m\mu = O(n)$. The simulator finally extracts Alice's input as X^*, a *subset* of $\tilde{X}$. Hence her input has size at most $n' = m\mu$.

However, the situation is likely slightly better than this strict upper bound. Looking closer, Alice can only send a set E of $n\mu$ (not $m\mu$) common encodings in the final step of the protocol. Each item $x \in \tilde{X}$ is associated with $\mu\mathsf{c}(x)$ common encodings, i.e. μ for each time she sends x in a $\mathcal{F}_{\mathsf{encode}}$ command as the receiver. So Alice is in the situation where if she wants more than n items to be represented in the set E, then at least one item must have one of its possible encodings excluded from E. This lowers the probability of that item being included in the final extracted input X^*.

In general, suppose for each $x \in \tilde{X}$, Alice includes $\mathsf{k}_i(x)$ encodings in her set E that are associated with the ith time she acted as $\mathcal{F}_{\mathsf{encode}}$ receiver with x. Hence $\sum_{x\in\tilde{X}}\sum_{i\in[\mathsf{c}(x)]}\mathsf{k}_i(x) \le n\mu$. Inspecting the simulation, we see that the probability a particular $x \in \tilde{X}$ survives to be included in X^* is $\Pr[x \in \tilde{X} \Rightarrow x \in X^*] = 1 - \prod_{\in[\mathsf{c}(x)]}(1 - \mathsf{k}_i(x))/\mu$ or simply $\mathsf{k}_1(x)/\mu$ in the case $\mathsf{c}(x) = 1$ (it happens only if the simulator happens to place x in a favorable position in the hash table). Hence, the expected size of X^* is $\sum_{x\in\tilde{X}}\sum_{i\in[\mathsf{c}(x)]}\mathsf{k}_i(x)/(\mu\mathsf{c}(x)) \le n$.

4.5.4 Encode-Commit Protocol

We now turn our attention to the encode-commit style PSI protocol described in Section 4.4.3 and outline how the optimizations of Section 4.5.1, 4.5.2 can be applied to it. Recall that the encode-commit protocol instructs Bob to encode his items as $\mathcal{F}_{\mathsf{encode}}$ receiver while Alice must send commitments of her items. The final step of this protocol is for Alice to send decommitments of her values encrypted under the corresponding $\mathcal{F}_{\mathsf{encode}}$ encodings.

It is straight forward to see that the hashing to bins technique of Section 4.5.1 is compatible with the encode-commit style PSI. When the optimization of aggregating masks across bins from Section 4.5.2 is applied, we observe that the situation becomes more complicated. Let us assume that Alice now sends the commitment to her value y together with the decommitment r encrypted under the encodings $\{[\![y']\!]^{\mathsf{A}}_{b,p} \mid p \in [\mu]\}$ where $(b, y') = \mathsf{phase}_{h,m}(y)$. That is, for a random order of $y \in Y$, Alice sends

$$\textsc{Comm}(y; r), \{[\![y']\!]^{\mathsf{A}}_{b,p} \oplus r \mid p \in [\mu]\}$$

to Bob. For each $x \in X$, Bob must trial decommit to all such $\textsc{Comm}(y; r)$ with the decommitment value $([\![y']\!]^{\mathsf{A}}_{b,\mathsf{p}(x)} \oplus r) \oplus [\![x']\!]^{\mathsf{A}}_{\mathsf{b}(x),\mathsf{p}(x)}$. This would result in Bob performing $O(n^2)$ trial decommitments, eliminating any performance benefits of hashing. This overhead can be reduced by requiring Alice to send additional information that allows Bob to quickly identify which decommitment to try. Specifically, we will use the $\mathcal{F}_{\mathsf{encode}}$ encodings to derive two values, $[\![v]\!]^{\text{TAG}}_{b,p} = \text{PRF}([\![v]\!]^{\mathsf{A}}_{b,p}, \text{TAG})$ and $[\![v]\!]^{\text{ENC}}_{b,p} = \text{PRF}([\![v]\!]^{\mathsf{A}}_{b,p}, \text{ENC})$. The important property here is that given the encoding $[\![v]\!]^{\mathsf{A}}_{b,p}$, both values can be derived, but without the encoding the two values appear pseudo-random and independent. We now have Alice send

$$\textsc{Comm}(y; r), \{[\![y']\!]^{\text{TAG}}_{b,p} \mid\mid ([\![y']\!]^{\text{ENC}}_{b,p} \oplus r) \mid p \in [\mu]\}$$

Bob can now construct a hash table mapping $[\![x']\!]^{\text{TAG}}_{b,p}$ to $([\![x']\!]^{\text{ENC}}_{b,p}, x)$. Upon receiving a commitment and the associated tagged decommitments, Bob can query each of Alice's tags in the hash table. If a match is found, Bob will add the associated x to the intersection if the associated $[\![x']\!]^{\text{ENC}}_{b,p}$ value is successfully used to decommits $\textsc{Comm}(y; r)$.

4.5.5 Encode-Commit Protocol Details & Security

We give a formal description of the protocol in Figure 4.4. The protocol requires a non-interactive commitment scheme. In Section 4.7 we discuss the security properties required of the commitment scheme. At a high level, we require an extractable commitment scheme with a standard (standalone) hiding requirement. In particular, we do not require equivocability. In the non-programmable random oracle, the standard scheme $H(x\|r)$ satisfies our required properties.

Theorem 5. *The protocol in Figure 4.4 is UC-secure in the $\mathcal{F}_{\mathsf{encode}}$-hybrid model, when the underlying commitment scheme satisfies Definition 6. The resulting protocol has cost $O(Cn \log n)$, where $C \approx \kappa$ is the cost of one (sender) $\mathcal{F}_{\mathsf{encode}}$ call on a $\sigma - \log n$ length bit string.*

Proof. Due to the similarity to the previous proof we defer giving hybrids and simply describe the simulators. We start with the case of a **corrupt Bob**. The simulator must extract Bob's input, and simulate the messages in the protocol. The simulator is nearly the same as in the previous protocol:

The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality. The simulator does nothing in Step 2. To extract Bob's set, the simulator observes all of Bob's $\mathcal{F}_{\mathsf{encode}}$ messages $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, p), y'_{b,p})$ in Step 3. The simulator computes $Y = \{\mathsf{phase}^{-1}_{h,m}(b, y'_{b,p}) \mid b \in [m], p \in [\mu]\}$ and sends it to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality which responds with the intersection $Z = X \cap Y$.

Set Z^* to be equal to Z along with arbitrary dummy items not in Y, so that $|Z^*| = n$. For each $z \in Z^*$, compute $(b, z') = \mathsf{phase}_{m,h}(z)$ and insert z' into bin $\mathcal{B}_X[b]$ at a random unused position $p \in [\mu]$. For $z \in Z^*$ in random order, compute $(b, z') = \mathsf{phase}_{m,h}(z)$ and send $\textsc{Comm}(z; r_z), \{[\![z']\!]^{\text{TAG}}_{b,p} \mid\mid [\![z']\!]^{\text{ENC}}_{b,p} \oplus r_z\}$ to Bob, where these encodings are obtained by playing the role of $\mathcal{F}_{\mathsf{encode}}$.

Importantly, the simulator extracts Bob's input in step 3 and thus knows the protocol output before step 4. It can therefore send appropriate commitments and use dummy commitments for those that are guaranteed not to be openable by Bob (those commitments whose decommitment values are perfectly masked by random encodings). Security follows from standard standalone hiding of the commitment scheme.

In the case of a **corrupt Alice** the simulator must simply extract Alice's effective input (Alice receives no output from $\mathcal{F}_{\mathsf{PSI}}$). The simulator is defined as follows:

The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality and initializes the commitment scheme in extraction mode (i.e., fixes the coin tossing in step 1 to generate simulated parameters). The simulator does nothing in Step 2 and Step 3. In Step 4, the simulator extracts Alice's commitments of the form $\textsc{Comm}(x; r_x)$ and inserts x' in the bin $\mathcal{B}_X[b]$ at a random unused position $p \in [\mu]$, where $(b, x') = \mathsf{phase}_{m,h}(x)$. Let S denote the set of the μ associated (*tag* || *decommit*) pairs. If there exists $(T \mid\mid D) \in S$ such that $T = [\![x']\!]^{\text{TAG}}_{b,p}$ and $\textsc{Comm}(x; r_x) = \textsc{Comm}(x; D \oplus [\![x']\!]^{\text{ENC}}_{b,p})$, add x to the set X^*. The simulator sends X^* to the $\mathcal{F}_{\mathsf{PSI}}$ functionality.

We see here that the simulator extracts candidate inputs for Alice by extracting from her commitments. Thus the protocol requires an extractable commitment scheme. This protocol also benefits from restricting Alice to a set of size exactly n item, unlike the dual execution protocol which achieves n items in exception and upper bounded by roughly $n' < 4n$ items. □

Set size n		2^8	2^{12}	2^{16}	2^{20}	2^{24}
LAN	μ	24	25	26	28	29
	m	64	1024	16384	262144	4194304
WAN	μ	40	43	45	47	49
	m	25	409	6553	104857	1677721

Table 4.1: Dual Execution hashing parameters μ, m for statistical security $\lambda = 40$.

4.5.6 Parameters

Let us now review the protocol as a whole and how to securely set the parameters. The parties first agree on hashing parameters that randomly map their sets of n items into m bins with the use of phasing. The bins are padded with dummy items to size $\mu = O(\log n)$. The parties both act as $\mathcal{F}_{\mathsf{encode}}$ receiver to encode all $m\mu$ items in their bins, including dummy items. Each bin position uses a unique $\mathcal{F}_{\mathsf{encode}}$ session. For all non-dummy encodings, both parties compute $\mu = O(\log n)$ common encodings. If an item is in the intersection, exactly one of these μ encodings will be the same for both parties. Alice then sends Bob all of these common encodings in a random order (not by bins). Bob is able to identify the matching encodings and infer the intersection.

By applying a bins into balls analysis, it can be seen that for m bins and n balls, the probability of there existing a bin with more than μ items is $\leq m\sum_{i=\mu+1}^{n}\binom{n}{i}\left(\frac{1}{m}\right)^{i}\left(1-\frac{1}{m}\right)^{n-i}$. Bounding this to be negligible in the security parameter gives the required bin size for a given n, m. By setting $m = O(n/\log n)$ and minimizing the overall cost, we obtain the set of parameters specified in Figure 4.1 with statistical security $\lambda = 40$. We found that $m = n/10$ minimizes the communication for our choices of n at the expense of increased computation when compared to $m = n/4$. As such, we choose $m = n/10$ in the WAN setting where communication is the dominant cost and $m = n/4$ in the LAN setting where computation has increased importance.

4.5.7 Discussion

Challenges of Two Party Output An obvious question is whether our protocol be extended to support two party output. In the semi-honest case, this is trivial, since the party who learns the intersection first can simply report it to the other. In the malicious setting, the parties cannot be trusted to relay this information faithfully.

A natural idea to solve this problem is to have Bob send all of his encodings to Alice, making the protocol completely symmetric. We briefly describe the problem with this approach. Suppose Bob behaves honestly with input set Y throughout most of the protocol. Let $y_0 \in Y$ be a distinguished element. In the last step, he sends his common encodings to Alice, but replaces all the encodings corresponding to y_0 with random values.

Now Bob will learn $X \cap Y$, but his effect on Alice will be that she learns only $X \cap (Y \setminus \{y_0\})$. More generally, a malicious Bob can always learn $X \cap Y$ but cause Alice to receive output $X \cap Y'$ for any $Y' \subseteq Y$ of Bob's choice.

4.6 Performance Evaluation

We have implemented several variants of out main protocol, and in this section we report on its performance. We denote our dual execution random-oracle protocol as DE-ROM and the encode-commit random-oracle protocol as EC-ROM. Only the dual execution protocol was implemented in the standard model and denoted as SM. We do not implement the encode-commit protocol in the standard model due to the communication overhead of standard model commitments such as [FJNT16], see 4.6.1 *Communication Cost*. All implementations are freely available at `github.com/osu-crypto/libPSI`.

We give detailed comparisons to two leading malicious-secure PSI protocols: our previous Bloom-filter-based protocol [RR17a] and the Diffie-Hellman-based protocol of De Cristofaro, Kim & Tsudik [DCKT10]. We utilized the implementation provided by [RR17a] of that protocol and [DCKT10]. All implementations were compared on the same hardware.

Implementation Details & Optimizations. We implemented our protocol in C++ and both the standard-model and random-oracle instantiation of $\mathcal{F}_{\mathsf{encode}}$, to understand the effect of the random-oracle assumption on performance.

We implement $\mathcal{F}_{\mathsf{encode}}$ by directly utilizing [OOS17] in the ROM model or with several chosen message 1-out-of-2 OTs [KOS15b] in the standard model

Setting	Protocol	Set size n										asymptotic
		2^8		2^{12}		2^{16}		2^{20}		2^{24}		
		Total	Online	Total	Online	Total	Online	Total	Online	Total	Online	
LAN	[KKRT16]*	0.19	0.19	0.21	0.21	0.4	0.4	3.8	3.8	59	59	$4n$ (RO)
	[DCKT10]	1.6	1.6	22.4	22.4	365	365	5630	5630	–	–	$6n$ (PK)
	[RR17a]	0.21	0.002	0.8	0.03	9.6	0.7	148	16	–	–	$4\kappa n$ (RO)
	Ours (EC-ROM)	0.13	0.004	0.19	0.06	0.94	0.69	12.6	11.3	239	218	$4n \log n$ (RO)
	Ours (DE-ROM)	0.13	0.006	0.23	0.08	1.3	1.0	18	16	296	261	
	Ours (SM, $\sigma = 32$)	0.15	0.018	0.48	0.19	3.5	1.8	56	31	–	–	$6\sigma n$ (CRH)
	Ours (SM, $\sigma = 64$)	0.19	0.034	0.84	0.31	8.0	3.7	134	35	–	–	
WAN	[KKRT16]*	0.56	0.56	0.59	0.59	1.3	1.3	7.5	7.5	107	106	$4n$ (RO)
	[DCKT10]	1.7	1.7	23.2	23.2	367	367	5634	5634	–	–	$6n$ (PK)
	[RR17a]	0.97	0.14	5.3	0.95	69	13	1080	216	–	–	$4\kappa n$ (RO)
	Ours (EC-ROM)	0.67	0.26	1.5	1.1	16	15	255	254	3208	3194	$4n \log n$ (RO)
	Ours (DE-ROM)	0.90	0.33	1.2	0.63	6.3	5.6	106	105	2647	2626	
	Ours (SM, $\sigma = 32$)	1.3	0.11	8.0	0.56	78	5.4	1322	115	–	–	$6\sigma n$ (CRH)
	Ours (SM, $\sigma = 64$)	1.9	0.14	16.8	0.74	226	82	3782	164	–	–	

Table 4.2: Single-threaded running time in seconds of our protocol compared to semi-honest [KKRT16] and malicious [DCKT10, RR17a]. We report both the total and online running time. DE-ROM, EC-ROM respectively denotes our dual execution and encode-commit model protocols. SM denotes the standard model dual execution variant on input bit length σ. Cells with – denote trials that either ran out of memory or took longer than 24 hours. (PK) denotes public key operations, (RO) denotes random oracle operations and (CRH) denotes correlation robust hash function operations. * [KKRT16] is a Semi-Honest secure PSI protocol. We show the [KKRT16] performance numbers here for comparison purposes.

as specified by Section 4.3. When we instantiate $\mathcal{F}_{\mathsf{encode}}$ with [OOS17], we use the BCH-$(511, 76, 171)$ linear code. As such, the $\mathcal{F}_{\mathsf{encode}}$ input domain is $\{0,1\}^{76}$. To support PSI over arbitrary length strings in the random-oracle model, we use the *hash to smaller domain* technique of [PSZ18] in conjunction with phasing. The hashed elements are 128 bits. This enables us to handle sets of size n such that $76 \geq \lambda + \log n$, e.g. $n = 2^{36}$ with $\lambda = 40$ bits of statistical security. For larger set sizes and/or security level, a larger BCH code can be used with minimal additional overhead. In the standard model, we perform PSI over strings of length 32 and 64 bits due to hash to smaller domain requiring the random-oracle to extract.

We used SHA1 as the underlying hash function, and AES as the underlying PRF/PRG (counter mode for a PRG) where needed. The random-oracle instantiation requires the OT-extension hash function to be modeled as a random-oracle. We optimize the $\mathcal{F}_{\mathsf{encode}}$ instantiations by not hashing dummy items.

The implementation of [DCKT10] uses the Miracl elliptic curve library using Curve 25519 achieving 128 bit computational security. It is in the random-oracle model and is optimized with the Fiat-Shamir sigma proofs. This implementation also takes advantage of the Comb method for fast exponentiation

Threads	Protocol	Set size n				
		2^8	2^{12}	2^{16}	2^{20}	2^{24}
4	[DCKT10]	0.61	6.9	95	1539	24948
	[RR17a]	0.15	0.52	5.8	84	–
	Ours (EC-ROM)	0.14	0.15	0.4	4.4	72
	Ours (DE-ROM)	0.14	0.17	0.6	7.0	93
	Ours (SM, $\sigma = 32$)	0.14	0.24	1.3	17	–
	Ours (SM, $\sigma = 64$)	0.15	0.37	2.7	40	–
16	[DCKT10]	0.33	2.2	29	458	7265
	[RR17a]	0.15	0.44	4.3	68	–
	Ours (EC-ROM)	0.14	0.16	0.4	3.0	42
	Ours (DE-ROM)	0.14	0.17	0.4	3.5	34
	Ours (SM, $\sigma = 32$)	0.14	0.18	0.6	7.5	–
	Ours (SM, $\sigma = 64$)	0.15	0.25	1.1	14.7	–
64	[DCKT10]	0.11	1.2	19	315	5021
	[RR17a]	0.14	0.34	2.1	32	–
	Ours (EC-ROM)	0.14	0.15	0.4	3.0	42
	Ours (DE-ROM)	0.14	0.17	0.4	2.9	25
	Ours (SM, $\sigma = 32$)	0.14	0.18	0.5	6.0	–
	Ours (SM, $\sigma = 64$)	0.15	0.21	1.0	14	–

Table 4.3: Total running times in seconds of our protocol compared to [DCKT10, RR17a] in the multi-threaded setting. Cells with – denote trials that ran out of memory.

	set size n					asymptotic	
	2^8	2^{12}	2^{16}	2^{20}	2^{24}	Offline	Online
[KKRT16]*	0.04	0.53	8	127	1956	$2\kappa^2$	$3n(\beta + \kappa)$
[DCKT10]	0.05	0.8	14	213	2356	0	$6n\phi + 6\phi + n\beta$
[RR17a]	1.9	23	324	4970	–	$2\kappa^2 + 2n\kappa^2$	$2n\kappa \log_2(2n\kappa) + n\beta$
Ours (EC-ROM)	0.29	4.8	79	1322	22038	$2\kappa^2$	$3\kappa n + n(C + D \log n + \log^2 n)$
Ours (DE-ROM)	0.25	3.5	61	1092	17875	$2\kappa^2$	$6\kappa n + \beta n \log n$
Ours (SM, $\sigma = 32$)	2.3	40	451	7708	–	$2\kappa^2 + 6\sigma\kappa n$	$\sigma n + \beta n \log n$
Ours (SM, $\sigma = 64$)	5.3	92	1317	22183	–		

Table 4.4: The empirical communication cost for both parties when configured for the WAN setting, listed in megabytes. Asymptotic costs are in bits. $\phi = 283$ is the size of the elliptic curve elements. $\beta \approx \lambda + 2\log n - 1$ bits is the size of the final masks that each protocol sends. $C \approx 2\kappa$ bits is the communication of performing one commitment and $D \approx \kappa$ is the size of a non-interactive decommitment.

(point multiplication) with the use of precomputed tables. The [DCKT10] protocol requires two rounds of communication over which $5n$ exponentiations and $2n$ zero knowledge proofs are performed. To increase performance on large set sizes, all operations are performed in a streaming manner, where data is sent as soon as it is ready.

The [RR17a] implementation is also highly optimized including techniques such as hashing OTs on demand and aggregating several steps in their cut and choose. To ensure a fair comparison, we borrow many of their primitives such as SHA1 and AES.

Experimental Setup. Benchmarks were performed on a server equipped with 2 multi-core Intel Xeon processors and 256GB of RAM. The protocol was executed with both parties running on the same server, communicating through the loopback device. Using the Linux `tc` command we simulated two network settings: a LAN setting with 10 Gbps and less than a millisecond latency; and a WAN setting with 40 Mbps throughput and 80ms round-trip latency.

All evaluations were performed with computational security parameter $\kappa = 128$ and statistical security $\lambda = 40$. We consider the sets of size $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$. The times reported are the average of 10 trials. Where appropriate, all implementations utilize the hardware accelerated AES-NI instruction set.

4.6.1 Results & Discussion

Execution time, single-threaded. Figure 4.2 shows the running time of our protocol compared with [DCKT10] and [RR17a] when performed with a single thread per party. We report both the total running time and the *online time*, which is defined as the portion of the running time that is input-dependent (i.e., the portion of the protocol that cannot be pre-computed).

Our experiments show that our ROM protocols' total running times are significantly less than the prior works, requiring 12.6 seconds to perform a set intersection for $n = 2^{20}$ elements in the LAN setting. A 11.7× improvement in running time compared to [RR17a] and a 447× improvement over [DCKT10]. Increasing the set size to $n = 2^{24}$, we find that our best protocol takes 239 seconds, whereas [RR17a] runs out of memory, and [DCKT10] requires over 24 hours. When considering the smallest set size of $n = 2^8$, our protocol remains the fastest with a running time of 0.13 seconds compared to 0.21 and 1.6 for [RR17a] and [DCKT10] respectively. Our standard model dual execution protocol is also faster than prior works when evaluated in the LAN setting, with a running time 2.6× faster than [RR17a] for $\sigma = 32$ and 1.1× faster for $\sigma = 64$.

Our ROM protocol also scales very well in the WAN setting where bandwidth and latency are constrained. For set size $n = 2^8$, all protocols require roughly 1 second with ours being slightly faster at 0.9 seconds. When increasing to a set size of $n = 2^{20}$ the difference becomes much more significant. Our DE-ROM protocol requires 106 seconds compared to 1080 for [RR17a], a 10× improvement. Our standard model protocol also has a fast online phase in the WAN setting due to the implementation moving a larger portion of the work to the offline as compared to the ROM protocol.

Multi-threaded performance. Figure 4.3 shows the total running times in the multi-threaded LAN setting. We see that our protocol parallelizes well, due to the fact that items are hashed into bins which can be processed more or less independently. By contrast [RR17a] uses a global Bloom filter representation for all items, which is less amenable to parallelization. For inputs of size $n = 2^{20}$ and 16 threads, our protocol is 23× faster than [RR17a] and 153× that of [DCKT10]. Increasing the number of threads from 1 to 16 speeds up our protocol by a factor of 5×, but theirs by a factor of only 2×.

While the Diffie-Hellman-based protocol of [DCKT10] is easily the most amenable to parallelization (16 threads speeding up the protocol by a factor of 12.3× for $n = 2^{20}$), its reliance on expensive public-key computations leaves it still much slower than ours.

Communication cost. Figure 4.4 reports both the empirical and asymptotic communication overhead of the protocols. The most efficient protocol with respect to communication overhead is [DCKT10]. The dominant term in their communication is to have each party send $3n$ field elements. The next most efficient is our DE-ROM protocol, requiring each party to send $O(n)$ encodings from $\mathcal{F}_{\mathsf{encode}}$. Concretely, for a set size of $n = 2^{20}$, our protocol requires 1.1 GB of communication, roughly 5× greater than [DCKT10]. However, on a modest connection of 40 Mbps, we find our protocol to remain the fastest even when [DCKT10] utilizes many threads. In addition, our protocol requires almost 5× less communication than [RR17a] (4.9GB).

When comparing our two ROM protocols, it can be seen that the dual execution technique requires less communication and is therefore faster in the WAN setting. The main overhead of the encode-commit protocol is the $O(n \log n)$ $\mathit{tag}||\mathit{decommitment}$ values that must be sent. This is of particular concern in the standard model where commitments are typically several times larger than their ROM counterparts. In contrast, the dual execution protocol sends $O(n \log n)$ encodings which can be less than half the size of a ROM decommitment.

One aspect of the protocols that is *not* reflected in the tables is how the communication cost is shared between the parties. In our DE-COM protocol, a large portion of the communication is in the encoding steps, which are entirely symmetric between the two parties. In [RR17a] the majority of the communication is done by the receiver (in the OT extension phase). Although the total communication cost of [RR17a] is roughly 5× that of our protocol, the communication cost to the receiver is $\sim 10\times$ ours.

Comparison with [RR17a]. We provide a more specific comparison to the protocol of Rindal & Rosulek [RR17a]. Both protocols are secure against malicious adversaries; both rely heavily on efficient oblivious transfers; neither protocol strictly enforces the size of a malicious party's input set (so both protocols realize the slightly relaxed PSI functionality of Figure 2.1).

We now focus on our random-oracle-optimized protocol, which uses the random-oracle instantiation of $\mathcal{F}_{\mathsf{encode}}$. As has been shown, this protocol is significantly faster than that of [RR17a]. We give a rough idea of why this should be the case. In [RR17a], the bulk of the cost is that the parties perform an OT for each bit of a Bloom filter. With n items, the size of the required Bloom filter is $\sim kn$, where k is the security parameter of the Bloom filter. For technical reasons, k in [RR17a] must be the computational security parameter of the protocol (*e.g.*, 128 in the implementation). Overall, roughly $\sim nk$ oblivious transfers are required.

The bulk of the cost in our protocol is performing the instances of $\mathcal{F}_{\mathsf{encode}}$. In our random-oracle instantiation, we realize $\mathcal{F}_{\mathsf{encode}}$ with the OT-extension protocol of [OOS17]. Each instance of $\mathcal{F}_{\mathsf{encode}}$ has cost roughly comparable to a plain OT. Our protocol requires $m\mu = O(n)$ such instances. It is this difference in the number of OT primitives that contributes the largest factor to the difference in performance between these two protocols.

We also observe that our standard model protocol is faster than [RR17a] in the LAN setting for $\sigma = 32$ and $\sigma = 64$. While it is true that [RR17a] only weakly depends on σ, it is still informative that our protocol remains competitive with the previous fastest protocol while eliminating the random-oracle assumption. When considering the WAN setting, the communication overhead of $\sigma m\mu = O(\sigma n)$ OTs limits our performance, resulting in $\sigma = 32$ being slightly slower than [RR17a].

Comparison with OPE protocols. Our protocol is orders of magnitude faster than blind-RSA based protocol of [DCKT10], due to [DCKT10] performing $O(n)$ exponentiations. Traditional OPE-based PSI also require $O(n)$ exponentiations and their running time would be similarly high. There are very recent OPE protocols based on OT but they still require $O(n)$ OTs plus $O(n/\kappa)$ relatively expensive interpolations of degree-$O(k)$ polynomials, totaling $O(n \log \kappa)$ operations. In contrast our protocol requires $O(n)$ OTs to be communicated and $O(n \log n)$ local OT computations.

Comparison with semi-honest PSI. An interesting point of comparison is to the state-of-the-art semi-honest secure protocol of Kolesnikov et al. [KKRT16] which follows the same PSZ paradigm. Figure 4.2 shows the running time of our protocol compared to theirs. For sets sizes up to $n = 2^{12}$ our protocol is actually faster than [KKRT16] in the LAN setting which we attribute to a more optimized implementation. Increasing the set size to $n = 2^{20}$ we see that our protocol require 12.6 seconds compared to 3.8 by [KKRT16], a 3.3× difference. For the largest set size of $n = 2^{24}$ we see the difference increase further to a 4× overhead in the LAN setting. In the WAN setting we see a greater difference of 25× which we attribute to the $\log n$ factor more communication/computation that our protocol requires.

4.7 Commitment Properties

The encode-commit variant of our protocol requires a non-interactive commitment scheme. The syntax is as follows:

- $\textsc{Setup}(1^\kappa)$: samples a random reference string crs.
- $\textsc{Comm}(crs, x, r)$: generates a commitment to x with randomness r. Note that in the main body, we omit the global argument crs.
- $\textsc{SimSetup}(1^\kappa)$: samples a reference string crs along with a trapdoor τ.
- $\textsc{Extract}(crs, \tau, c)$: extracts the committed plaintext value from a commitment c.

We require the scheme to satisfy the following security properties:

Definition 6. *A commitment scheme is secure if the following are true:*

1. *(Extraction:) Define the following game:*

```
$\mathsf{ExtractionGame}(1^\kappa, \mathcal{A})$:
    $(crs, \tau) \leftarrow \textsc{SimSetup}(1^\kappa)$
    $(c, x', r') \leftarrow \mathcal{A}(crs)$
    if $c = \textsc{Comm}(crs, x', r')$ and $x' \neq \textsc{Extract}(crs, \tau, c)$:
        return 1
    else: return 0
```

   The scheme has straight-line extraction if for every PPT $\mathcal{A}$, $\mathsf{ExtractionGame}(1^\kappa, \mathcal{A})$ outputs 1 with negligible probability.

2. *(Hiding:) Define the following game:*

```
$\mathsf{HidingGame}(1^\kappa, \mathcal{A}, b)$:
    $crs \leftarrow \textsc{Setup}(1^\kappa)$
    $(x_0, x_1) \leftarrow \mathcal{A}(crs)$
    $r \leftarrow \{0,1\}^\kappa$
    return $\textsc{Comm}(crs, x_b, r)$
```

   The scheme is hiding if, for all PPT $\mathcal{A}$, the distributions $\mathsf{HidingGame}(1^\kappa, \mathcal{A}, 0)$ and $\mathsf{HidingGame}(1^\kappa, \mathcal{A}, 1)$ are indistinguishable.

The definitions are each written in terms of a single commitment, but they apply simultaneously to many commitments using a simple hybrid argument.

In the non-programmable random oracle model, the classical commitment scheme $\textsc{Comm}(x, r) = H(x\|r)$ satisfies these definitions. In the standard model, one can use any UC-secure non-interactive commitment scheme, e.g., the efficient scheme of [FJNT16].

4.8 Formal Encode-Commit Protocol

Parameters: X is Alice's input, Y is Bob's input, where $X, Y \subseteq \{0,1\}^\sigma$. m is the number of bins and μ is a bound on the number of items per bin. The protocol uses instances of $\mathcal{F}_{\mathsf{encode}}$ with input length $\sigma - \log n$, and output length $\lambda + 2\log(n\mu)$, where λ is the security parameter.

1. **[Parameters]** Parties agree on a random hash function $h : \{0,1\}^\sigma \to [m]$ using a coin tossing protocol.

2. **[Hashing]**

   (a) For $x \in X$, Alice computes $(b, x') = \mathsf{phase}_{h,m}(x)$ and adds x' to bin $\mathcal{B}_X[b]$ at a random unused position $p \in [\mu]$.

   (b) For $y \in Y$, Bob computes $(b, y') = \mathsf{phase}_{h,m}(y)$ and adds y' to bin $\mathcal{B}_Y[b]$ at a random unused position $p \in [\mu]$.

   Both parties fill unused bin positions with the zero string.

3. **[Encoding]** For bin index $b \in [m]$ and position $p \in [\mu]$:

   (a) Let x' be the value in bin $\mathcal{B}_X[b]$ at position p. Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, b, p), x')$ to the $\mathcal{F}_{\mathsf{encode}}$ functionality which responds with $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, b, p), [\![x']\!]^{\mathsf{B}}_{b,p})$. Bob receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, b, p))$ from $\mathcal{F}_{\mathsf{encode}}$.

   (b) Let y' be the value in bin $\mathcal{B}_Y[b]$ at position p. Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, p), y')$ to the $\mathcal{F}_{\mathsf{encode}}$ functionality which responds with $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, p), [\![y']\!]^{\mathsf{A}}_{b,p})$. Alice receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, p))$ from $\mathcal{F}_{\mathsf{encode}}$.

4. **[Output]**

   (a) **[Alice's Common Mask]** For each $x \in X$, in random order, let b, p be the bin index and position that x' was placed in during Step 2a to represent x. For $j \in [\mu]$, Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, j), x')$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, j), [\![x']\!]^{\mathsf{A}}_{b,j})$ in response. Alice sends

   $$[\![x']\!]^{\mathsf{A}}_{b,j} \oplus [\![x']\!]^{\mathsf{B}}_{b,p}$$

   to Bob. Let E denote the $n\mu$ encodings that Alice sends.

   (b) **[Bob's Common Mask]** Similarly, for $y \in Y$, let b, p be the bin index and position that y' was placed in during Step 2b to represent y. For $j \in [\mu]$, Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, b, j), y')$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, b, j), [\![y']\!]^{\mathsf{B}}_{b,j})$ in response. Bob outputs

   $$\left\{ y \in Y \;\middle|\; \exists j \in [\mu] : [\![y']\!]^{\mathsf{A}}_{b,p} \oplus [\![y']\!]^{\mathsf{B}}_{b,j} \in E, \text{ where } (b, y') = \mathsf{phase}_{h,m}(y) \right\}$$

Figure 4.3: Our malicious-secure Dual Execution PSI protocol.

Parameters: X is Alice's input, Y is Bob's input, where $X, Y \subseteq \{0,1\}^\sigma$. m is the number of bins and μ is a bound on the number of items per bin. The protocol uses instances of $\mathcal{F}_{\mathsf{encode}}$ with input length $\sigma - \log n$, and output length $\lambda + 2\log(n\mu)$, where λ is the security parameter.

1. **[Parameters]** Parties agree on a random hash function $h : \{0,1\}^\sigma \to [m]$ and global parameters for the commitment scheme, using a coin tossing protocol.

2. **[Hashing]**

   (a) For $x \in X$, Alice computes $(b, x') = \mathsf{phase}_{h,m}(x)$ and adds x' to bin $\mathcal{B}_X[b]$ at a random unused position $p \in [\mu]$.

   (b) For $y \in Y$, Bob computes $(b, y') = \mathsf{phase}_{h,m}(y)$ and adds y' to bin $\mathcal{B}_Y[b]$ at a random unused position $p \in [\mu]$.

   Both parties fill unused bin positions with the zero string.

3. **[Encoding]** For bin index $b \in [m]$ and position $p \in [\mu]$: Let y' be the value in bin $\mathcal{B}_Y[b]$ at position p. Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, p), y')$ to the $\mathcal{F}_{\mathsf{encode}}$ functionality which responds with $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, p), \llbracket y' \rrbracket^{\mathsf{A}}_{b,p})$. Alice receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, p))$ from $\mathcal{F}_{\mathsf{encode}}$. Bob computes

   $$\llbracket y' \rrbracket^{\textsc{tag}}_{b,p} = \mathrm{PRF}(\llbracket y' \rrbracket^{\mathsf{A}}_{b,p}, \textsc{tag})$$
   $$\llbracket y' \rrbracket^{\textsc{enc}}_{b,p} = \mathrm{PRF}(\llbracket y' \rrbracket^{\mathsf{A}}_{b,p}, \textsc{enc})$$

   and constructs a hash table H mapping $\llbracket y' \rrbracket^{\textsc{tag}}_{b,p}$ to $(\llbracket y' \rrbracket^{\textsc{enc}}_{b,p}, y)$.

4. **[Output]** For each $x \in X$, in random order, let b, p be the bin index and position that x' was placed in during Step 2a to represent x. For $j \in [\mu]$, Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, b, j), x')$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, b, j), \llbracket x' \rrbracket^{\mathsf{A}}_{b,j})$ in response. For each response Alice computes $\llbracket x' \rrbracket^{\textsc{tag}}_{b,j} = \mathrm{PRF}(\llbracket x' \rrbracket^{\mathsf{A}}_{b,j}, \textsc{tag})$ and $\llbracket x' \rrbracket^{\textsc{enc}}_{b,j} = \mathrm{PRF}(\llbracket x' \rrbracket^{\mathsf{A}}_{b,j}, \textsc{enc})$.

   For each x Alice sends the tuple

   $$\textsc{Comm}(x; r_x), \quad \{\llbracket x' \rrbracket^{\textsc{tag}}_{b,j} \mid\mid \llbracket x' \rrbracket^{\textsc{enc}}_{b,j} \oplus r_x \mid j \in [\mu]\}$$

   to Bob who outputs the union of all y such that $\exists j : \llbracket x' \rrbracket^{\textsc{tag}}_{b,j} \in H.keys$ and $\textsc{Comm}(x, r_x) = \textsc{Comm}(y; (\llbracket x' \rrbracket^{\textsc{enc}}_{b,j} \oplus r_x) \oplus \llbracket y' \rrbracket^{\textsc{enc}}_{*})$ where $(\llbracket y' \rrbracket^{\textsc{enc}}_{*}, y) := H[\llbracket x' \rrbracket^{\textsc{tag}}_{b,j}]$.

Figure 4.4: Our Encode-Commit PSI protocol.

Chapter 5

PSI with Differential Privacy

Cheaper Private Set Intersection via Differentially Private Leakage by Adam Groce, Peter Rindal & Mike Rosulek

5.1 Introduction

Secure (two-party) computation allows two parties to evaluate a function on private inputs, and learn only the output of the function. Standard security definitions for secure computation provide extremely strong cryptographic guarantees. In many situations it is reasonable to consider relaxing these guarantees if such relaxation results in a significantly faster protocol.

In this work, we consider relaxing security by allowing the protocol to leak some "extra" information. This raises the question, what extra information is reasonable to leak without completely undermining the idea of secure computation? A natural candidate is to leak only *differentially private* (DP) information about the private inputs. Differential privacy [DMNS06] captures the idea that only *aggregate* information about a data-set is leaked, but not information that is specific to any single individual in the data-set. Beimel et al. [BNO08] were the first to suggest allowing differentially private leakage in secure computation protocols, by replacing the standard indistinguishability notion (in the privacy requirement) with a corresponding differential privacy statement. In this work, we will consider a slightly different security definition that considers malicious adversaries.

5.1.1 Chapter Contributions

We show a significant tradeoff between performance and differentially private leakage in the setting of private set intersection (PSI). In PSI, two parties have input sets X and Y, and wish to learn (only) their intersection $X \cap Y$. Differential privacy is an especially good match for this problem, since many of its most notable applications involve sets of *individuals.* For example, the application of private contact discovery in social networks involves a client with a small set of email addresses (i.e., address book), and a server with a large set of user email addresses, who compute the corresponding intersection. Differential privacy in this setting would ensure that, e.g., the server doesn't learn information about any single person in the client's address book.

Our starting point is the protocol of Rindal & Rosulek [RR17b], which is the fastest malicious-secure PSI protocol to date. Like many PSI protocol, this one works by first having the parties hash their items into bins (e.g., item x is placed in bin $h(x)$), and a smaller PSI is performed in each bin.

In all such PSI protocols, dummy items must be added to fill each bin to a maximum size. This is because the number of true items in each bin is information that cannot be inferred from the intersection alone (i.e., it is leakage from a security standpoint). Unfortunately, dummy items vastly outnumber real items, and inflate the protocol's cost (relative to what would be needed for correctness alone).

The load of each bin can be thought of as a *histogram* of a party's input set. Instead of *completely* hiding this histogram (by padding with dummy items to an upper bound), we can consider releasing it in a differentially-private manner since histograms are one of the canonical statistics that work well with DP. If the differentially private histogram is closer to the true histogram, it will result in significantly less dummy items and a more efficient protocol.[1] We explore two specific approaches:

(1) The standard way to release a histogram with differential privacy is to add modest Laplacian noise to each value (i.e., load of each bin). However, correctness in our setting is broken if the parties *underestimate* the load of a bin. We introduce new methods for computing a "differentially-private overestimate" of the true histogram, taking advantage of the known (balls in bins) distribution of items.

[1] In this high-level overview we focus on the private histogram technique. However, there is also another aspect of the baseline PSI protocol which can be improved with differentially private leakage, which we discuss later.

(2) Another mechanism is to identify a small set of threshold values and always round up the true number of items in each bin to the closest threshold value. One can think of the fully-secure protocol as rounding up all bin loads to a maximum value B, and the simplest generalization would include another intermediate threshold t (now parties treat each bin as having either t or B items). This mechanism leads to even more efficient PSI, and provides a guarantee known as *distributional differential privacy (DDP)* [BGKS13]. The DDP model requires there to be sufficient uncertainty in the sensitive data, which may not be present in all PSI application scenarios. We fully discuss the applicability of DDP leakage in PSI in Section 5.6.2.

We have incorporated our new techniques into the implementation of [RR17b] and present a thorough analysis of our performance improvements. Allowing differentially private leakage improves the performance of the protocol by 27% (for $\epsilon = 0.5$) to 46% (for $\epsilon = 4$). Allowing $\epsilon = .33$ distributionally DP leakage for Alice (and $\epsilon = 1$ standard DP for the receiver) improves the performance by 15%.

5.1.2 Related Work with Leakage

Trading leakage for performance in generic MPC The dual execution paradigm, introduced by Mohassel & Franklin [MF06a] and extended in other work [HKE12, KMRR15], relaxes the malicious security model by allowing the adversary to learn an arbitrary *bit* about the honest party's input. The resulting protocol is only 2× the cost of a standard semi-honest protocol. The dual execution paradigm can be used to securely evaluate any functionality, but does not guarantee that the leakage satisfies any meaningful limitation (other than being 1 bit). Our approach also relaxes security by allowing some leakage, but in the specific case of PSI we ensure that the leakage is differentially private.

Beimel et al. [BNO08] explored a model of MPC where the standard privacy guarantee is replaced by the natural differentially private analog. This security model is inherently tied to the semi-honest adversarial setting, and not suitable for our malicious setting. We believe our security model more clearly separates the different intuitive security goals of MPC and makes it explicit which goals are ensured in a cryptographic or differentially private sense (e.g., we enforce correctness in the standard sense but input independence and privacy in a differentially private sense).

Trading leakage for performance in special-purpose MPC Another notable instance of trading leakage for performance is in the context of searchable encryption, and more generally secure database search. The encrypted database systems BlindSeer [PKV$^+$14] and that of Cash et al. [CJJ$^+$13] provide standard database (DBMS) functionality, while hiding the nature of the query from the server with the exception of some leaked information. Schoppmann et al. [SGB18] perform nearest neighbor queries on a private database, while allowing differentially private leakage. As in our work, this leakage is tailored to the protocol and specific functionality.

Oblivious RAM (ORAM) [Gol87, Ost90] refers to a protocol between a client and server that allows the client to read/write its data held by the server, but hiding the client's access pattern from the server. Some relatively recent work [WCM16, MG17, CCMS17] has explored relaxing the obliviousness requirement of ORAM to allow the server to learn only differentially private information about the access pattern. Similar to our work, the goal is to compute a "plain" functionality with the standard notion of correctness — i.e., without adding any noise. In this case, the functionality corresponds to RAM read/write instructions; in our case the functionality is PSI.

In all works in this section, the security model is honest-but-curious.

Securely Computing Differentially-Private Functions There is a great deal of work on using secure computation to evaluate a differentially-private mechanism – e.g., [PNH17, KOV14, CGBL$^+$17, RN10, BSMD10] – including protocols for closely related functionalities like cardinality of set intersection cardinality [NH12] and union [FMJS17]. In these works, the focus is to compute a differentially-private functionality, under a *standard security notion*. In other words, the parties learn *only* a differentially private answer, while in our work the parties learn a non-differentially private function, plus some *additional* (incidental) differentially private information.

5.2 Background

5.2.1 Differential Privacy

Differential privacy [DMNS06] is a condition meant to ensure that releases from large databases do not violate the privacy of any of the individuals whose data that database contains. In particular, let $X = (X_1, X_2, \ldots, X_n)$ be a database of information about n individuals, with X_i being the data corresponding to the i^{th} individual. A query is a (possibly randomized) function f that takes a database as input. Differential privacy ensures that the inclusion of individual i in the database does not allow an adversary to infer anything about them.

The strength of the definition is controlled by a parameter ϵ. Intuitively, differential privacy guarantees that whatever $f(X)$ outputs, that output would have been roughly equally likely even if any individual had instead changed their data arbitrarily. If the observed output was roughly equally likely for any value of the individual's data, then the adversary cannot infer anything having seen it. (This intuition has been formalized. See [KS08] for a rigorous treatment.)

More formally, we say that two databases X and X' are *neighboring* if they differ only in the addition or deletion of a single value. We can then define differential privacy.

Definition 7. *A query function f is ϵ-differential private if for any two neighboring databases X and X' and for any set S of possible outputs, we have*

$$\frac{\Pr[f(X) \in S]}{\Pr[f(X') \in S]} \leq e^{\epsilon}. \tag{5.1}$$

Differential privacy also benefits from composition. That is, if f is ϵ_1-differentially private and g is ϵ_2-differentially private, the function h that gives both their outputs (i.e., $h(X) = (f(X), g(X))$) is $(\epsilon_1 + \epsilon_2)$-differentially private. This is useful for two reasons. First, it's a useful tool for building differentially private functions, since one can build them up from smaller private components. Second, it means that the definition protects people's privacy even if many separate differentially private comptuations are done, including queries called by different people.

Differential privacy is also preserved by any additional post-processing. That is, if f is ϵ-differentially private, then so is $g \circ f$ for any g. This means anything that can be computed from a private output (with additional access to the database) is itself private. This, like composition, is useful both because it's an intuitively desirable property of a privacy definition and because it's useful in practice when designing private queries.

Differentially private queries have been created that approximate a wide variety of naturally desirable output. In the most simple query algorithms, the output is simply equal to the desired (non-private) output plus some carefully calibrated noise. In particular, say that F is some function on databases with output in $\mathbb{R}^n$. We define the *sensitivity* F to be the maximum amount of change F can see when one row of the database is changed.

Definition 8. *The* sensitivity *of a function F with output in $\mathbb{R}^n$ is* $\max ||F(X)-F(X')||$*, where the maximum is taken over all neighboring databases X and X' and the norm is an ℓ_1 norm.*

Our techniques rely primarily on several uses of the Laplace mechanism. This uses random noises generated according to a *Laplace distribution*, a double-sided exponential distribution.

Definition 9. *The* Laplace distribution Lap_c*, is parameterized by a scaling parameter c. We denote the probability density function at x with $\mathsf{Lap}_c(x)$. Precisely,*

$$\mathsf{Lap}_c(x) = \frac{1}{2c} e^{-|x|/c}. \tag{5.2}$$

Any function can be made differentially private by adding Laplace noise proportional to its sensitivity and $1/\epsilon$. (In some cases this is optimal, while in others more elaborate techniques can give greater utility.)

Theorem 10. *Let F be a function with sensitivity ΔF and output in $\mathbb{R}^n$. Let f be a randomized algorithm with output $f(X) = F(X) + \mathsf{Lap}^n_{\Delta F/\epsilon}$, where $\mathsf{Lap}^n_{\Delta F/\epsilon}$ denotes a vector of n independently chosen values from the Laplace distribution with parameter $c = \Delta F/\epsilon$. Then f is ϵ-differentially private.*

5.3 Security model

We introduce a model for secure computation (secure function evaluation) in which the protocol allows differentially private leakage.

Other papers have defined and explored notions of MPC with differentially private leakage (e.g., [BNO08, SGB18]). However, these works are in the semi-honest model which is arguably much simpler. In that model it is possible to simply insist that the adversary's view is a differentially private function of the honest parties' inputs. It is also common to consider protocols that compute (differentially private) approximations of some natural function.

Parameters:

- Functions $f_1, f_2 : (\{0,1\}^*)^2 \to \{0,1\}^*$
- Class of leakage function pairs $\mathcal{L}$

Behavior:
If no parties are corrupt:

1. Wait for inputs x_1 from P_1 and x_2 from P_2
2. Deliver output $f_1(x_1, x_2)$ to P_1 and $f_2(x_1, x_2)$ to P_2

If any party is corrupt, let P_c denote the corrupt party and P_h denote the honest party. Then:

1. Wait for input x_h from P_h and $(\textsc{leak}, L_{\mathsf{pre}})$ from P_c, where $\mathcal{L}$ contains some pair of the form $(L_{\mathsf{pre}}, \cdot)$.
2. Give $L_{\mathsf{pre}}(x_h)$ to P_c
3. Wait for inputs x_c and $(\textsc{leak}, L_{\mathsf{post}})$ from P_c, where $(L_{\mathsf{pre}}, L_{\mathsf{post}}) \in \mathcal{L}$.
4. Give $L_{\mathsf{post}}(x_h)$ and $f_c(x_1, x_2)$ to P_c
5. Wait for input $(\textsc{deliver}, b)$ from P_c. If $b = 0$ deliver output $\perp$ to P_h; otherwise if $b = 1$ deliver output $f_h(x_1, x_2)$ to P_h.

Figure 5.1: Ideal functionality $\mathcal{F}_{\mathsf{leak},f,\mathcal{L}}$ for securely evaluating function $f = (f_1, f_2)$ with leakage.

On the other hand, our work is in the malicious adversarial model and we are interested in *exactly* computing a natural function, but allowing extra differentially private leakage. We found it most natural to define security using traditional simulation-based security, but weakening the ideal functionality to provide additional, *explicit* leakage on the honest party's inputs to the adversary. As we will see, leakage in the malicious model also has the potential to affect the *input independence* requirement for secure computation (a problem which is not present in the semi-honest model).

Figure 5.1 describes an ideal functionality for securely evaluating a function while also giving some leakage. The adversary's choice of leakage is constrained to some class $\mathcal{L}$ of allowable leakage functions (a parameter of the functionality). The functionality allows the corrupt party to obtain leakage on the honest party's input at two different times: both before and after choosing its own input. Hence, both the privacy and *input independence* guarantees of secure computation are degraded in a differentially-private way. I.e., a corrupt party cannot make its choice of input significantly depend on any *individual record* in the honest party's set. However, in this model correctness is preserved in the standard sense.

Definition 11. *A protocol π securely realizes $f = (f_1, f_2)$* ***with $\mathcal{L}$ leakage*** *if π is a UC-secure protocol for $\mathcal{F}_{\mathsf{leak},f,\mathcal{L}}$ (Figure 5.1).*

The protocol realizes f ***with ϵ-DP leakage*** *if it realizes f with $\mathcal{L}$ leakage, where for every $(L_{pre}, L_{post}) \in \mathcal{L}$, the function $x \mapsto (L_{pre}(x), L_{post}(x))$ is ϵ-differentially private. (Here we assume some "neighbor" relation for the possible values of x).*

An interesting special case of this model is one where $\mathcal{L}$ contains only pairs of the form $(L_{\mathsf{pre}}, \perp)$ or of the form $(\perp, L_{\mathsf{post}})$, where $\perp$ is overloaded to denote the function $\perp(x) = \perp$ for all x. This special case corresponds to the setting where the corrupt party receives only one phase of leakage. In particular, if the L_{pre} leakage is $\perp$, then input independence is guaranteed in the standard way (i.e., not degraded in a differentially private manner).

5.4 PSI Protocol Framework

In this section we describe our PSI protocol framework (i.e., PSI with histogram leakage). The framework is based on the PSI protocol of Rindal & Rosulek [RR17b], which is the fastest known PSI protocol achieving malicious security.

5.4.1 Overview

We first review the Rindal-Rosulek protocol. At a high level, the protocol works by having the parties first hash their items into bins (with a random hash function). Within each bin, they perform a quadratic-cost PSI protocol on the items assigned to that bin. However, the abstraction boundary is broken slightly by an optimization that combines together some information across all of the bins.

Parameters: two parties denoted as Sender and Receiver. The input domain $\{0,1\}^\sigma$ and output domain $\{0,1\}^\ell$ for a private F.

1. **[Initialization]** Create an initially empty associative array $F : \{0,1\}^\sigma \to \{0,1\}^\ell$.
2. **[Receiver Encode]** Wait for a command $(\textsc{Encode}, \mathsf{sid}, c)$ from the Receiver, and record c. Then:
3. **[Adversarial Map Choice]** If the sender is corrupt, then send $(\textsc{RecvInput}, \mathsf{sid})$ to the adversary and wait for a response of the form $(\textsc{Deliver}, \mathsf{sid}, F_{\mathsf{adv}})$. If the sender is honest, set $F_{\mathsf{adv}} = \bot$. Then:
4. **[Receiver Output]** If $F_{\mathsf{adv}} = \bot$ then choose $F[c]$ uniformly at random; otherwise set $F[c] := F_{\mathsf{adv}}(c)$, interpreting F_{adv} as a circuit. Give $(\textsc{Output}, \mathsf{sid}, F[c])$ to the receiver. The n:
5. **[Sender Encode]** Stop responding to any requests by the receiver. But for any number of commands $(\textsc{Encode}, \mathsf{sid}, c')$ from the sender, do the following:
   - If $F[c']$ doesn't exist and $F_{\mathsf{adv}} = \bot$, choose $F[c']$ uniformly at random.
   - If $F[c']$ doesn't exist and $F_{\mathsf{adv}} \neq \bot$, set $F[c'] := F_{\mathsf{adv}}(c')$.
   - Give $(\textsc{Output}, \mathsf{sid}, c', F[c'])$ to the sender.

Figure 5.2: The Oblivious Encoding ideal functionality $\mathcal{F}_{\mathsf{encode}}$ [OOS17]

Quadratic PSI This step of the protocol uses an oblivious encoding functionality, which can be thought of as an *oblivious pseudorandom function* (OPRF) or a variant of random oblivious transfer over an exponentially large number of values. The functionality chooses a random function which we denote as $x \mapsto [\![x]\!]$. A receiver can learn $[\![x]\!]$ for a *single, chosen* value x, while the sender can compute $[\![x]\!]$ for *any number of* values x. Concretely, the protocol uses the OT extension protocol of Orrù, Orsini & Scholl [OOS17], which is secure in the random oracle model. It achieves a slight relaxation of this functionality where a corrupt sender is allowed to choose the mapping $[\![\cdot]\!]$. This variant functionality is described formally in Figure 5.2, and it is sufficient for use in the PSI protocol.

If Alice has n items $\{x_1, \ldots, x_n\}$ and Bob has n items $\{y_1, \ldots, y_n\}$, they can use this oblivious encoding functionality to perform PSI in the following way:

1. For each x_j, the parties perform an instance of the oblivious encoding with Alice acting as receiver, so that Alice learns $[\![x_j]\!]^{\mathsf{B}}_j$. We use $[\![\cdot]\!]^{\mathsf{B}}_j$ to refer to the encoding that is known to Bob.

2. Symmetrically, for each y_i, the parties perform an oblivious encoding instance in the other direction, with Bob learning $[\![y_i]\!]^{\mathsf{A}}_i$.

3. The parties define n^2 "common encodings" of the form:

$$[\![v]\!]^{*}_{i,j} \overset{\text{def}}{=} [\![v]\!]^{\mathsf{A}}_i \oplus [\![v]\!]^{\mathsf{B}}_j$$

The idea of the common encodings is that Alice can predict the value of $[\![v]\!]^{*}_{i,j}$ *if and only if* she used v as input to the encoding step involving $[\![\cdot]\!]^{\mathsf{B}}_j$. Bob can predict the value iff he used v as input to the encoding $[\![\cdot]\!]^{\mathsf{A}}_i$.

The protocol continues with Alice sending the (randomly ordered) set of encodings:

$$E = \{[\![x_j]\!]^{*}_{i,j} \mid i,j \in [n]\}$$

Bob can check this set for encodings that are known to him (i.e., all encodings of the form $[\![y_i]\!]^{*}_{i,j}$). Bob computes the protocol output as:

$$Z = \{y_i \mid \exists j : [\![y_i]\!]^{*}_{i,j} \in E\}$$

Suppose the encodings $[\![\cdot]\!]$ have length $\lambda + 2\log_2 n$ bits. Then the probability of $[\![x_j]\!]^{*}_{i,j} = [\![y_i]\!]^{*}_{i,j}$ for $x_j \neq y_i$ is bounded by $2^{-\lambda}$. Conditioned on this event not happening, the protocol is correct. We defer a discussion of this protocol's security for later.

Hashing and Dummy Items This basic protocol requires only $2n$ instances of the oblivious encoding functionality (n in each direction), but requires Alice to communicate n^2 encodings.

A standard approach to reduce the complexity of a PSI protocol (dating back at least to [FNP04]) is for parties to first choose[2] a random hash function $h : \{0,1\}^* \to [m]$ and use this function to assign their items into m bins. Then the quadratic-cost protocol can be performed on the items within each bin.

[2] For security reasons, h should be chosen by a secure coin-tossing protocol.

As mentioned in Section 5.1.1, it is necessary to hide the number of items per bin — in the ideal world, it is not possible to infer the number of items that honest party has that satisfy $h(x) = b$ for a chosen bin b. To obscure this information, the parties add dummy items to each bin until each bin has exactly the worst-case number of items (such a bound can be computed using standard balls-in-bins analysis).

A typical choice of parameters is $m = O(n/\log n)$ bins. In that case, both the *expected* and *worst case* (with overwhelming probability) number of items per bin is $O(\log n)$. Then the overall cost of the protocol is $m \cdot O(\log^2 n) = O(n \log n)$.

However, further optimization is still possible. Let μ be the (padded) size of each bin. As currently described, Alice would send μ encodings for each of her items, *including her dummy items!* However, it is public information that *overall* Alice has only n items and hence only $n\mu$ encodings corresponding to them. Only the distribution of these dummies *within* the bins is secret. So the suggestion of [RR17b] is to let Alice gather all $n\mu$ non-dummy encodings from all bins, shuffle them, and send them together. Now the encodings must be somewhat longer (the probability of a spurious collision in these encodings has increased), and Bob must lookup candidate encodings in a larger set, rather than in small bin-specific sets. However, the gain in communication makes this optimization an overall improvement to the protocol.

5.4.2 Our Generalization and Details

As mentioned in Section 5.1.1, our modification of the Rindal-Rosulek protocol is to release some differentially private information about the number of items in each bin. We begin by defining the properties we require:

Definition 12. *Let $h : \{0,1\}^* \to [m]$ be a hash function. For a set of items X hashed into m bins by h, the load of bin i is:*

$$\mathsf{Ld}_h(X, i) = \#\{x \in X \mid h(x) = i\},$$

The load of all bins can be written as a vector:

$$\mathsf{Ld}_h(X) = \Big(\mathsf{Ld}_h(X, 1), \ldots, \mathsf{Ld}_h(X, m)\Big)$$

Definition 13. *For two vectors $u = (u_1, \ldots, u_m)$ and $v = (v_1, \ldots, v_m)$, write $u \le v$ if $u_i \le v_i$ for all $i \in [m]$. Let $\widetilde{L}_h$ be a randomized function. We call $\widetilde{L}$*

*a **load overestimate** if for all X*

$$\Pr[\mathsf{Ld}_h(X) \leq \widetilde{L}_h(X)] \text{ is overwhelming,}$$

where the probability is taken over the randomness of $\widetilde{L}$ and the choice of h.

One can think of the function $\widetilde{L}_h(X) = (\mu, \dots, \mu)$ to be the load-overestimate used in [RR17b], where μ is an upper bound on bin size computed using balls-in-bins analysis.

Our generalization of Rindal-Rosulek is parameterized by a load overestimate $\widetilde{L}$. Roughly speaking, after choosing the hash function h:

- Alice hashes her items X into bins, computes the load of each bin $(a_1, \dots, a_n) = \mathsf{Ld}_h(X)$, and an overestimate $(\widetilde{a}_1, \dots, \widetilde{a}_m) \leftarrow \widetilde{L}_h(X)$. She adds dummy items to each bin until the load (including dummy items) equals the overestimate.
- Likewise, Bob hashes his items Y into bins. He can compute an overestimate $(\widetilde{b}_1, \dots, \widetilde{b}_m) \leftarrow \widetilde{L}_h(Y)$, however [RR17b] security proof / simulation breaks down in the case where Alice has more items in a bin than Bob (we discuss this fact in Appendix 5.5). We must therefore have Bob compute $\widetilde{c}_i = \max\{\widetilde{a}_i, \widetilde{b}_i\}$ and add dummy items so that the load in the bins is $(\widetilde{c}_1, \dots, \widetilde{c}_m)$.[3]
- Within each bin i, the parties perform the standard quadratic PSI between $\widetilde{a}_i$ and $\widetilde{c}_i$ $(\geq \widetilde{a}_i)$ items.

In [RR17b], it is public information that Alice will have $n\mu$ total non-dummy encodings, since each bin contains exactly μ items. In our case, Alice will have $\sum a_i \widetilde{c}_i$ non-dummy encodings. She cannot simply send the non-dummy encodings as in [RR17b] since the number of these items is indeed sensitive to her true bin-load. To protect this information, we need a differentially-private overestimate of this inner product:

Definition 14. *We say that randomized function $\widetilde{I}$ is a **inner-product overestimate** if for all vectors $u = (u_1, \dots, u_m)$ and $v = (v_1, \dots, v_m)$,*

$$\Pr[\langle u, v \rangle \leq \widetilde{I}(u, v)] \text{ is overwhelming}$$

[3] While it would be better for Bob to use $\widetilde{b}_1, \dots, \widetilde{b}_m$ as the load-overestimate, our experiments show that it only would improve performance by a few percent.

In our generalization, Alice computes a differentially-private[4] inner-product overestimate $\widetilde{e} = \widetilde{I}((a_1, \ldots, a_m), (\widetilde{c}_1, \ldots, \widetilde{c}_m))$. She includes the non-dummy encodings in a set E, adds random values until $|E| = \widetilde{e}$, and finally sends the (permuted) contents of E to Bob.

The formal details of the protocol are given in Figure 5.3. Unsurprisingly, the security proof is extremely similar to that of [RR17b]. For the sake of completeness, we present the self-contained proof in Section 5.5.

Theorem 15. *The protocol securely realizes the leaky PSI functionality $\mathcal{F}_{\mathsf{leak,PSI},\mathcal{L}}$, for leakage $\mathcal{L}$ where every $(L_{\mathsf{pre}}, L_{\mathsf{post}}) \in \mathcal{L}$ has the form:*

$$\begin{aligned} L_{\mathsf{pre}}(S) &= \widetilde{L}_h(S) \\ L_{\mathsf{post}}(S) &= \begin{cases} \widetilde{I}(\mathsf{Ld}_h(S), (\widetilde{c}_1, \ldots, \widetilde{c}_m)) & \text{if Bob corrupt} \\ \bot & \text{if Bob honest} \end{cases} \end{aligned}$$

(i.e., for some $h, \widetilde{c}_1, \ldots, \widetilde{c}_m$).

Roughly speaking, the only differences from [RR17b] are that (1) the number of items per bin is changed to be input-dependent; (2) the number of common encodings sent by Alice is input-dependent. However, if the simulator is given these values (e.g., as leakage from the functionality), then its simulation proceeds just as in [RR17b].

Adversary's set size. The protocol of Rindal & Rosulek does not strictly enforce the number of items in the adversary's set. Roughly speaking, instead of using dummy items, an adversary can choose to use actual items in their place. Hence, instead of being limited to using n items, the adversary might use as many as $\sum_i \widetilde{a}_i$ items. This fact is reflected in the ideal functionality (Figure 2.1), where an honest party provides a set of n items while a corrupt party can provide a set of $n' > n$ items for some bound n'. The analysis of the n' bound in [RR17b] applies here: briefly, $n' = O(n)$ with a small hidden constant factor.

[4] Specifically, for all second arguments, $\widetilde{I}$ should be a differentially private function of its first input.

5.5 Security Proof

In this section we prove the security of the (leaky) PSI protocol (Figure 5.3). The proof is essentially the same as the one in [RR17b], and is presented here with many aspects identical to [RR17b]. The modifications have to do with the leakage handling in the simulator, which is highlighted below.

Proof. We start with the case of a **corrupt Bob**. The simulator must extract Bob's input, and simulate the messages in the protocol. We first describe the simulator:

> The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality. The simulator obtains leakage $(\widetilde{a}_1, \ldots, \widetilde{a}_m) \leftarrow L_{\mathsf{pre}}(X) = \widetilde{L}_h(X)$ from the functionality and simulates this as the message from Alice in Step 2a. It receives $(\widetilde{c}_1, \ldots, \widetilde{c}_m)$ from corrupt Bob in Step 2b. To extract Bob's set, the simulator observes all of Bob's $\mathcal{F}_{\mathsf{encode}}$ messages $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, d, p), y_{d,p})$ in Step 3b. The simulator computes $Y = \{y_{d,p}\}$ and sends it to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality which responds with the intersection $Z = X \cap Y$.
>
> For each $z \in Z$, compute bin index $d = h(z)$ and place z into a random unused position $p \in [\widetilde{a}_d]$ in bin $\mathcal{B}_X[z]$. For $j \in [\widetilde{c}_d]$, add value $\llbracket z \rrbracket^{\mathsf{A}}_{d,p} \oplus \llbracket z \rrbracket^{\mathsf{B}}_{d,j}$ to a set E. Then obtain leakage $\widetilde{e} \leftarrow L_{\mathsf{post}}(X) = \widetilde{I}(\mathsf{Ld}_h(X), (\widetilde{c}_1, \ldots, \widetilde{c}_m))$ from the functionality and pad E with random values until it has size $\widetilde{e}$. The simulator then sends E (randomly permuted) to the adversary.

To show that this is a valid simulation, we consider a series of hybrids.

Hybrid 0 The first hybrid is the real interaction where Alice honestly uses her input X, and $\mathcal{F}_{\mathsf{encode}}$ is implemented honestly.

Observe Bob's commands to $\mathcal{F}_{\mathsf{encode}}$ of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, d, p), y_{d,p})$ in Step 3b. Based on these, we can define (but not use) the set $\tilde{Y} = \{y_{d,p}\}$.

Hybrid 1 In this hybrid, we modify Alice to send dummy values to $\mathcal{F}_{\mathsf{encode}}$ in Step 2a (rather than her actual inputs). The hybrid is indistinguishable by the properties of $\mathcal{F}_{\mathsf{encode}}$.

Hybrid 2 In Step 4a, for each $x \in X$ the simulated Alice sends common encodings of the form $\llbracket x \rrbracket^{\mathsf{A}}_{d,j} \oplus \llbracket x \rrbracket^{\mathsf{B}}_{d,p}$, for some position p, where $d = h(x)$. Suppose $x \notin \tilde{Y}$. By construction of $\tilde{Y}$, Bob never obtained an encoding of the form $[\![x]\!]^{\mathsf{A}}_{d,j}$. This encoding is therefore distributed independent of everything else in the simulation. In particular, the *common* encoding of the form $[\![x]\!]^{\mathsf{A}}_{d,j} \oplus [\![x]\!]^{\mathsf{B}}_{d,p}$, which is added to the set E, is uniform.

We therefore modify the hybrid in the following way. In Step 4a, simulated Alice generates encodings for the set E only for items in the intersection $Z = X \cap \tilde{Y}$, instead of X as before. She still pads the set E to contain $\widetilde{e}$ items, as before. By the above argument, the adversary's view is identically distributed in this modified hybrid.

We can see that the final hybrid uses the contents X only in the following way: It uses the bin-load overestimate $\widetilde{L}_h(X)$; it uses the result of $X \cap \tilde{Y}$ for some set $\tilde{Y}$ that it computes; it uses the inner-product overestimate $\widetilde{I}(\mathsf{Ld}_h(X), (\widetilde{c}_1, \ldots, \widetilde{c}_m))$. Hence, this hybrid corresponds to our final simulator, where we obtain leakage from the functionality, send $\tilde{Y}$ to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality and receive $X \cap \tilde{Y}$ in response.

We now turn our attention to a **corrupt Alice**. In this case the simulator must simply extract Alice's effective input (Alice receives no output from $\mathcal{F}_{\mathsf{PSI}}$). The simulator is defined as follows:

The simulator plays the role of the ideal $\mathcal{F}_{\mathsf{encode}}$ functionality. The simulator obtains $(\widetilde{a}_1, \ldots, \widetilde{a}_m)$ from the adversary in Step 2a and obtains leakage $(\widetilde{b}_1, \ldots, \widetilde{b}_m) \leftarrow L_{\mathsf{pre}}(Y) = \widetilde{L}_h(Y)$ from the functionality. It computes $(\widetilde{c}_1, \ldots, \widetilde{c}_m)$ as in the protocol and simulates this as the message from Bob in Step 2b. In Step 3a, the simulator intercepts Alice's commands of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, d, p), x_{d,p})$. The simulator computes a set of candidates $\tilde{X} = \{x_{d,p}\}$ and for $x \in \tilde{X}$ let $\mathsf{c}(x)$ denote the multiplicity of x in its bin; i.e., the number of values p for which $x = x_{h(x),p}$.

The simulator computes a hash table $\mathcal{B}$ as follows. For $x \in \tilde{X}$ the simulator places $\mathsf{c}(x)$ copies of x in bin $\mathcal{B}[h(x)]$. Note that we place $\sum_{x:h(x)=d} \mathsf{c}(x)$ items in bin index d. By construction $\sum_{x:h(x)=d} \mathsf{c}(x) \leq \widetilde{a}_d \leq \widetilde{c}_d$, and hence simulated Bob has enough space in bin d. Let the items in each bin be randomly permuted, and for each x, let $\mathsf{p}(x)$ denote the set of positions of x in its bin.

Let E denote the set of values sent by Alice in Step 4a. The simulator computes

$$X^* = \big\{x \in \tilde{X} \mid \exists j \in [\widetilde{c}_{h(x)}], p \in \mathsf{p}(x) : \\ [\![x]\!]^{\mathsf{A}}_{h(x),j} \oplus [\![x]\!]^{\mathsf{B}}_{h(x),p} \in E\big\} \tag{5.3}$$

where the encodings are obtained by playing the role of $\mathcal{F}_{\mathsf{encode}}$. The simulator sends X^* to the $\mathcal{F}_{\mathsf{PSI}}$ functionality.

Hybrid 0 The first hybrid is the real interaction where Bob honestly uses his input X, and $\mathcal{F}_{\mathsf{encode}}$ is implemented honestly.

Observe Alice's commands to $\mathcal{F}_{\mathsf{encode}}$ of the form $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, d, p), x_{d,p})$ in Step 3a. Based on these, define $\tilde{X} = \{x_{d,p}\}$.

Hybrid 1 In this hybrid, we modify Bob to send dummy inputs to $\mathcal{F}_{\mathsf{encode}}$ in Step 2b (rather than his actual items). The hybrid is indistinguishable by the properties of $\mathcal{F}_{\mathsf{encode}}$.

Hybrid 2 Note that in this hybrid, Bob's output is computed as specified in Step 4c. We then modify the interaction so that Bob removes all output items which are not in $\tilde{X}$. The hybrids differ only in the event that simulated Bob computes an output in Step 4c that includes an item $y \notin \tilde{X}$. This happens only if $[\![y]\!]^{\mathsf{A}}_{d,j} \oplus [\![y]\!]^{\mathsf{B}}_{d,p} \in E$, where Bob places y in position p of bin $d = h(y)$. Since $y \notin \tilde{X}$, however, the encoding $[\![y]\!]^{\mathsf{B}}_{d,p}$ is distributed uniformly. The length of encodings is chosen so that the overall probability of this event (across all choices of $y \notin \tilde{X}$) is at most $2^{-\lambda}$. Hence the modification is indistinguishable.

Hybrid 3 We modify the hybrid in the following way. When building the hash table, the simulated Bob uses $\tilde{X}$ instead of his actual input Y. Each $x \in \tilde{X}$ is inserted with multiplicity $\mathsf{c}(x)$. Then he computes the protocol output as specified in Step 4c; call it X^*. This is not what the simulator gives as output — rather, it gives $X^* \cap Y$ as output instead.

The hashing process is different only in the fact that items of $Y \setminus \tilde{X}$ are excluded and replaced in the hash table with items of $\tilde{X} \setminus Y$ (*i.e.*, items in $Y \cap \tilde{X}$ are treated exactly the same way). Note that the definition of $\tilde{X}$ ensures that the hash table can hold all of these items without overflowing. Also, this change is local to Step 4c, where the only thing that happens is Bob computing his output. However, by the restriction added in *Hybrid 2* , items in $Y \setminus \tilde{X}$ can never be included in X^*. Similarly, by the step added in this hybrid, items in $\tilde{X} \setminus Y$ can never be included in the simulator's output. So this change has no effect on the adversary's view (which includes this final output).

The final hybrid works as follows. A simulator interacts with the adversary and at some point computes a set X^*, without the use of Y. Then the simulated Bob's output is computed as $X^* \cap Y$. Hence, this hybrid corresponds to our final simulator, where we send X^* to the ideal $\mathcal{F}_{\mathsf{PSI}}$ functionality, which sends output $X^* \cap Y$ to ideal Bob. □

5.6 Choosing Appropriate Leakage

In this section we try to private load overestimates and inner-product overestimates that will allow for faster protocol execution. We begin with differential privacy. We then consider distributional differential privacy.

5.6.1 Differentially private leakage

Let h be the hash function that is chosen in the malicious protocol. Alice and Bob both hash each element of their input using h and divide them into m bins. In the protocol of Rindal and Rosulek, an upper bound is chosen such that with overwhelming probability no bin will exceed that bound, and bins are padded with dummy values up to that upper bound. (This is needed because the true bin sizes leak information about the input.) As a result, in practice most elements being compared are dummy items. Our goal here is to find tighter bounds on the bin sizes so that fewer dummy items can be used.

Let a and b be vectors of bin counts, with a_i being the number of Alice's input items that hashed to bin i (and analogously with b_i and Bob's input). These vectors a and b are essentially histograms, and we can release differentially private estimates of histograms using known techniques. Histograms have a sensitivity of 1, so by Theorem 10 we can add noise from $\mathsf{Lap}_{1/\epsilon_1}$ to each entry and achieve ϵ_1-differential privacy.

Now, we cannot simply set $\widetilde{a}_i = a_i + \mathsf{Lap}_{1/\epsilon_1}$ because the Laplace distribution includes negative numbers and $\widetilde{a}_i$ would not be an overestimate. We could instead set $\widetilde{a}_i = a_i + \mathsf{Lap}_{1/\epsilon_1} + z$ for some constant z chosen such that the probability that $\mathsf{Lap}_{1/\epsilon_1} < -z$ is negligible. This trick has been used before, for example by Kellaris et al. (in a different setting) [KKNO17].

Unfortunately, this overestimate mechanism gives only marginal efficiency improvements. In practice, most bins are reported to have the worst-case maximum load (i.e., the same as in fully-secure [RR17b]) under this mechanism. We refine this mechanism by taking into account not only the differentially private estimate of bin size, but also the *prior knowledge* Alice and Bob have. In particular, they know that the protocol uses an ideal hash function that places each input into each bin with equal probability. As a result, for each input there is an (independent) $1/m$ chance that that input ends up in bin i, meaning that the size of each bin follows a binomial distribution. Let $\mathsf{Binom}_{n,p}(x)$ be the probability that a binomial distribution with n objects and a success probability p for each object takes value x. For now we omit subscripts, since they are always n and $p = 1/m$.

We want Alice and Bob to use the differentially private bin size estimates not to *replace* their binomial distribution priors, but rather to *update* them. This is a simple Bayes' Theorem calculation. Let $\hat{a}_i$ and $\hat{b}_i$ be the differentially private bin estimate (i.e., $\hat{a}_i = a_i + \mathsf{Lap}_{1/\epsilon_1}$). Let $\mathsf{Post}(x|t)$ be the probability that $a_i = x$ (or $b_i = x$) given that $\hat{a}_i = t$ (or $\hat{b}_i = t$). We can compute $\mathsf{Post}(x|t)$ as follows:

$$\begin{aligned}\mathsf{Post}(x|t) = &\quad \Pr[a_i = x | \hat{a}_i = t] \\ = &\quad \frac{\Pr[\hat{a}_i = t | a_i = x] \Pr[a_i = x]}{\sum_y (\Pr[\hat{a}_i = x | a_i = y] \Pr[a_i = y])} \\ = &\quad \frac{\mathsf{Lap}_{1/\epsilon_1}(t-x)\mathsf{Binom}(x)}{\sum_y (\mathsf{Lap}_{1/\epsilon_1}(x-y)\mathsf{Binom}(y))}\end{aligned}$$

This expression is hard to handle analytically, but it can easily be computed. Experimental measurements of computational efficiency of PSI protocols generally have n in the range of 2^{12} to 2^{24}, and the optimal value of m in this range is generally between $n/4$ and $n/10$. In this range, $\mathsf{Binom}(y)$ becomes negligible when $y > 100$, so the sum in the denominator of the above expression can be computed quite easily.[5] As a result, it is not hard to compute, for a given $\hat{a}_i$ and a given potential value of $\widetilde{a}_i$, what the probability is that $a_i > \widetilde{a}_i$.

Given a desired failure probability of $2^{-\lambda}$, we choose $\widetilde{a}_i$ such that $\Pr[a_i > \widetilde{a}_i] < 2^{-\lambda}/m$, meaning that a union bound implies that with all but $2^{-\lambda}$ probability we have $a_i \leq \widetilde{a}_i$ for all i. The same is done for $\widetilde{b}_i$. Note that because $\widetilde{a}_i$ is computed from $\hat{a}_i$ with no additional access to the private data, the secure post-processing property of differential privacy guarantees that it is also a private output (and therefore acceptable to leak).

[5] As we discuss later, the modifications we make actually make the optimal number of bins smaller, but not by enough to make this computation difficult.

Histogram estimate $\hat{a}_i$	Resulting bound $\widetilde{a}_i$
-2	21
2	21
4	22
6	23
10	25
14	28
30	38

Table 5.1: Examples of inferred bounds given particular estimates of bin size. For these calculations, $n = 2^{2}0$, $m = n/4$, and $\epsilon = 1$. Because the average bin size is 4, the most common estimate is also 4, so most bins are roughly size 22, compared to 31 in RR17.

An example of the results of this calculation is shown in Figure 5.1. We note that the estimate does allow for a significantly smaller bound. Interestingly, the properties of the Laplace distribution result in negative estimates for bin size all being equivalent, with smaller values not resulting in lower bounds on the size of the bin. (In contrast, very large estimates do result in very large bounds, even higher than in the no-leakage protocol. However, estimates that large are negligibly likely to occur.) The table shows only integer values, but of course the $\hat{a}_i$ estimate can be any real number. In practice we compute a table with values at some chosen density and then the protocol rounds the estimate up to the closest value where the bound has been precomputed.

We must also upper bound the number of masks that Alice and Bob must share. The ideal minimal value here is $I = \sum_i a_i \widetilde{b}_i$, but this cannot be released exactly. We instead again release a differentially private estimate (this time with parameter ϵ_2) and use it to compute an upper bound. This estimate is computed and released by Alice, taking $\widetilde{b}_i$ as a fixed constant. So the sensitivity of d is $\Delta I = \max_i\{\widetilde{b}_i\}$. With this, Alice can release the private estimate $\hat{I} = I + \mathsf{Lap}_{\Delta d/\epsilon_2}$.

As before, $\hat{I}$ could easily be too low. However, this time instead of doing Bayesian inference we use the more simple technique discussed and ruled out for the earlier bound. We simply set the upper bound $\widetilde{I}$ equal to $\hat{I} + z$, for an appropriate constant z. As long as the noise from $\mathsf{Lap}_{\Delta I/\epsilon_2}$ is less than $-z$, $\widetilde{I}$ is a valid upper bound. The cumulative density function at z of $\mathsf{Lap}_{\Delta I/\epsilon_2}$ (for negative z) is $0.5e^{\|z\epsilon_2/\Delta I\|}$, so we simply choose z such that this value is $2^{-\lambda}$.

This allows a fully secure protocol with leakage that is ϵ-differentially private with $\epsilon = \epsilon_1 + \epsilon_2$ (due to the composition theorem discussed earlier). The total allowed privacy budget ϵ can be divided between the two stages so as to maximize efficiency. It seems clear theoretically that one would want to allocate the vast majority of the privacy budget to the $\widetilde{a}$ and $\widetilde{b}$ queries, letting ϵ_2 be quite small in comparison to ϵ_1, but the precise values we use are chosen through experimental optimization.

5.6.2 Distributional differential privacy

We feel it is worth considering the release of even more precise information about the histogram of values. Say the receiver was to take the histogram of hash values and announce for each bin whether or not it was larger than the average bin size n/m. This would allow removing most of the dummies in roughly half of the bins, but it is a deterministic query so can't possibly be differentially private (for any ϵ). For example, if Alice had sufficiently specific side information about the receiver's input to know that there were at least n/m items in one bin, but the receiver said there were no more than this, then Alice could rule out all other values that mapped to that bin. But that would require knowing with significant probability several of the receiver's input values, and not just any values — specifically values that mapped to the same bin under a random hash function chosen at runtime. There are certainly settings where one party might already know most of the other party's input, but there are also settings where such side information is implausible. Generally speaking, attacks that take advantage of this kind of leakage seem unlikely.

If one is working in a use case where this sort of side information is unlikely, then we can use more nuanced notions of privacy to give better efficiency. In particular, we consider here *distributional differential privacy (DDP)*, introduced by Bassily et al. [BGKS13].[6] Rather than considering the worst case over all possible databases, DDP treats the database itself as a random variable, modeling adversarial uncertainty. This means that a given proof that a mechanism is DDP only holds for a particular set of possible adversarial knowledge.

[6] DDP is a special case of *coupled-worlds privacy*. Coupled-worlds privacy has additional parameters and allows for more flexibility in choosing the most relevant privacy definition for a particular use case. Here we use DDP because it is the most straightforward instantiation, the one we feel is most well-motivated here, and the one that is most analogous to differential privacy.

Definition 16. *Say Δ is a set of distributions for (X, Z), where X is the database and Z is the adversary's auxiliary information about the database. A mechanism f has $(\epsilon, \delta, \Delta)$*-distributional differential privacy *if for all distributions $\mathcal{D} \in \Delta$, all i, all (x_i, z) in the support of (X, Z), and all possible sets of output Q,*

$$\begin{aligned} &\Pr[f(X) \in Q \mid X_i = x_i, Z = z] \\ &\qquad \leq e^\epsilon \Pr[f(X_{-i}) \in Q \mid X_i = x_i, Z = z] + \delta \end{aligned}$$

and

$$\begin{aligned} &\Pr[f(X_{-i}) \in Q \mid X_i = x_i, Z = z] \\ &\qquad \leq e^\epsilon \Pr[f(X) \in Q \mid X_i = x_i, Z = z] + \delta, \end{aligned}$$

where X_i is the i^{th} item of the database and X_{-i} is the database with that row removed.[7]

Inuitively, the definition guarantees to an individual that, given the adversary's present knowledge/uncertainty, the mechanism will output roughly the same distribution of values whether or not that individual's data is included in the database. As a result, the inclusion of this data does not allow the adversary is unable to learn anything (except a small amount parameterized by ϵ) about the value. The definition is less stringent than differential privacy, more narrowly tailored to require the minimal thing necessary to maintain privacy. But this is not without cost. If the adversary knows more about the database than Δ allows, then there is no guarantee of privacy. (Fortunately, if the adversary knows *less* there is no problem.) Additionally, DDP does not have automatic composition in the way that differential privacy does, so it cannot safely be combined with other private data releases without a case-by-case analysis. The nuances of privacy definitions are quite subtle, and we refer the reader to [BGKS13] for a more detailed discussion.

5.6.3 PSI with DDP leakage

We begin with the most favorable possible setting. We assume (for now) that Alice knows nothing at all about the hashes of the receiver's inputs.[8] Note

[7] The original DDP definition uses a simulator $\mathsf{Sim}(X_{-i})$ in place of $F(X_{-i})$. In all cases here we will choose $\mathsf{Sim} = F$, so we present a simplified definition.

[8] Formally, Δ consists of all distributions on (X, Z) such that conditioned on any value z in the support of Z, the distribution on X is such that with high probability of the

that this does not mean that the receiver's inputs are completely unknown, just that they have sufficient uncertainty that their hashes under the random hashing function appear random. Note also that this is an assumption we can make regarding *only Alice's knowledge* of Bob's input, because the PSI protocol itself gives output to Bob, which is auxiliary information about Alice's input that violates this assumption.

Our private mechanism is simple. We choose a given threshhold t and then for each bin i release (with complete accuracy) whether $\mathsf{Ld}_h(X, i) < t$ or $\mathsf{Ld}_h(X, i) \geq t$. This means that for all small bins, dummies can be added only up to a size of t. Formally, the information that will be leaked in this setting is $q(X) = S$, where S is the set of all indices of bins with load t or greater.

We now wish to compute the privacy parameters ϵ and δ for this mechanism. We first want to fix a given output S and consider the following ratio:

$$\rho = \frac{\Pr[q(X) = S \mid h(X_i) = j]}{\Pr[q(X_{-i}) = S \mid h(X_i) = j]} \tag{5.4}$$

We note first of all that the calculation will be the same for all choices of i and j, so we assume without loss of generality that $i = n$ and $j = m$. We also leave the conditioning implicit, since from this point forward all probabilities are conditioned on $h(X_n) = m$. We now want to examine the same ratio, with the following simplified notation:

$$\rho = \frac{\Pr[q(X) = S]}{\Pr[q(X_{-n}) = S]} \tag{5.5}$$

We first consider the case where $m \in S$, i.e. the case where $\rho \geq 1$. Let $S_{-m} = S \setminus \{m\}$ and let C_m be a random variable equal to the size of bin m. We can then rewrite the numerator.

$$\begin{aligned} \Pr[q(X) = S] = &\Pr[q(X_{-n}) = S] \\ &+ \Pr[q(X_{-n}) = S_{-m} \wedge C_m = t-1] \\ = \Pr[q(X_{-n}) = S]& \\ + \Pr[q(X_{-n}) = S_{-m} &\mid C_m = t-1] \Pr[C_m = t-1] \end{aligned} \tag{5.6}$$

We now define a random variable $H_{n,m}$ equal to the set of indices of bins of size t or greater when n balls are thrown into m bins. For example,

choice of h, each $h(X_i)$ is a uniform, independent value.

$q(X_{-n}) = H_{n-1,m}$ and once we condition on $C_m = t-1$, we have $q(X_{-n}) = H_{n-t,m-1}$. (To see this, note that it's clear when we condition on a particular set of $t-1$ values ending up in bin m, and the overall probability is the average of the value when conditioned on each specific set.) We also note that $\Pr[C_m = t-1] = \mathsf{Binom}_{n-1,1/m}(t-1)$. As a result, we have

$$\begin{aligned}\Pr[q(X) = S] =& \Pr[H_{n-1,m} = S] \\ &+ \Pr[H_{n-t,m-1} = S_{-m}] \cdot \mathsf{Binom}_{n-1,1/m}(t-1).\end{aligned}$$

Note that the denominator in Equation 5.5 can also be written this way, with $\Pr[q(X_{-n}) = S] = \Pr[H_{n-1,m} = S]$. Putting these two simplifications together, we get

$$\rho = 1 + \mathsf{Binom}_{n-1,1/m}(t-1)\frac{\Pr[H_{n-t,m-1} = S_{-m}]}{\Pr[H_{n-1,m} = S]}. \tag{5.7}$$

We then note that all sets S of a given size are equally likely values of H (for any choice of subscripts), so setting $s = |S|$ we can rewrite the fraction in the above equation as

$$\frac{\Pr[H_{n-t,m-1} = S_{-m}]}{\Pr[H_{n-1,m} = S]} = \frac{\Pr[|H_{n-t,m-1}| = s-1]/\binom{m-1}{s-1}}{\Pr[|H_{n-1,m}| = s]/\binom{m-1}{s}}.$$

The binomial expressions then simplify:

$$\begin{aligned}\frac{\binom{m-1}{s}}{\binom{m-1}{s-1}} &= \frac{(m-1)!/[s!(m-s-1)!]}{(m-1)!/[(s-1)!(m-s)!]} \\ &= \frac{(s-1)!(m-s)!}{s!(m-s-1)!} = (m-s)/s\end{aligned}$$

That leaves us with the following expression for the ratio ρ:

$$\rho = 1 + \mathsf{Binom}_{n-1,1/m}(t-1) \cdot \frac{m-s}{s} \cdot \frac{\Pr[|H_{n-t,m-1}| = s-1]}{\Pr[|H_{n-1,m}| = s]}$$

We must then also consider the case where $m \notin S$ and $\rho < 1$. The analysis of the denominator is unchanged, but the numerator must be handled separately in this case. Luckily it proceeds quite similarly. We have

$$\begin{aligned}\Pr[q(X) = S] = &\Pr[q(X_{-n}) = S] \\ &- \Pr[q(X_{-n}) = S \wedge C_m = t - 1].\end{aligned}$$

This is almost the same as in Equation 5.6, except that the addition has changed to subtraction and in the second term we have S instead of S_{-m}. Doing the same manipulation gives us

$$\rho = 1 - \mathsf{Binom}_{n-1,1/m}(t-1)\frac{\Pr[H_{n-t,m-1} = S]}{\Pr[H_{n-1,m} = S]}. \tag{5.8}$$

As before we use the symmetry of $\Pr[H_{n-t,m-1} = S]$ across all sets S of the same size. Again using $|S| = s$ we have

$$\frac{\Pr[H_{n-t,m-1} = S]}{\Pr[H_{n-1,m} = S]} = \frac{\Pr[|H_{n-t,m-1}| = s]/\binom{m-1}{s}}{\Pr[|H_{n-1}| = s]/\binom{m-1}{s}}.$$

Unlike in the previous case, these binomial coefficients cancel and we are left with

$$\rho = 1 - \mathsf{Binom}_{n-1,1/m}(t-1) \cdot \frac{\Pr[|H_{n-t,m-1}| = s]}{\Pr[|H_{n-1,m}| = s]}.$$

Estimating concrete parameters If we knew how to compute an exact probability distribution for $|H_{n,m}|$ we would now be finished. The distribution of $|H_{n,m}|$ approximates a normal distribution. The value of ρ is worst when s is far from its expected value, so we could simply define an interval $[\alpha, \beta]$ and compute the values of ρ with $s = \alpha$ and $s = \beta$ and set $\epsilon = \max|\ln(\rho)|$. That would bound the ratio for all values in $[\alpha, \beta]$ and we could then compute the probability that $s \notin [\alpha, \beta]$ and set δ to that value. By adjusting α and β we could control a tradeoff between ϵ and δ.

Unfortunately we cannot find an efficiently computable closed-form expression for the distribution of $|H_{n,m}|$, and the normal approximation, while pretty good, is unreliable for very precise estimates of the extreme tails needed for δ calculations. So we instead resort to experimental estimation of these values. We run a simulation of throwing $n - t$ (resp. $n - 1$) balls into $m - 1$ (resp. m) bins many times, each time noting the number of bins ending up with at least t balls.

Concretely, computational limitations meant we could only run a large simulation for a single choice of parameters, and not for the largest values of n.

We choose $n = 2^{16}$, $m = n/10 = 6553$, and $t = 13$ (meaning that each bucket is said to be either in $[0, 12]$ or $[12, \infty)$). We were able to run a simulation with 20 million data points. On average 1366 bins will be over the threshold, and our data gave us confidence to say that by choosing $[\alpha, \beta] = [1257, 1473]$ we achieve delta of approximately $1/500,000$. In principle ϵ should be worst near the extreme tails, but we found that as we moved to the extreme tails we hit a point where our data was too sparse to make accurate estimates before we hit a point where ϵ was increasing. In the entire range where we have sufficient data, variation in ϵ is minimal and the maximum value it takes is 0.33. This is a lower value that we were using in the differentially private protocol, so we should ideally accept a worse ϵ value in exchange for an even better δ value, but our data is insufficient to quantify the tradeoff beyond this point.

We note also that while simulations with larger n will take even more computational power, it is clear that the privacy parameters will only improve, as the distribution of $|H_{n,m}|$ is still approximately normal but now with a larger standard deviation and therefore a better ratio between adjacent values. We therefore consider the protocol with $m = n/10$ and $t = 13$ to be $(0.33, 5 \times 10^{-5})$-DDP in the $n = 2^{20}$ case as well. In reality, the parameters are probably significantly better than this.

Because we saw no sign of increasing ϵ in the range of values we could estimate, we believe these estimates are in fact much worse than what is really achieved. A successful theoretical analysis of these parameters, rather than relying on empirical estimation, would be a highly desirable result of future work.

Relaxing the assumption on adversarial knowledge When the adversary does in fact know some information about some of the values in the input set, the DDP privacy of the mechanism is not completely compromised. Knowing some values outside the bin in question is similar in nature to decreasing n by that amount. Knowing values in the same bin as X_i (and i is arbitrary) is similar to reducing t by that amount. That means privacy will degrade rapidly when the adversary knows (or has significant certainty about) several values that would all hash to the same bin under the random function h. The chances of this are low unless the adversary knows a large amount about the input set. However, while we are confident of these general statements, quantifying the privacy guarantees with a specific choice of adversarily knowledge would (given the techniques discussed here) require

			LAN Setting				WAN Setting			
n	Protocol	ϵ	n/m	Total Time (ms)	Online Time (ms)	Comm. (MB)	n/m	Total Time (ms)	Online Time (ms)	Comm. (MB)
2^{12}	This	4	14	216	73	2.2	14	1129	622	2.24
		2	12	217	73	2.6	16	1196	685	2.54
		1	12	216	73	3.0	16	1274	768	2.95
		0.5	12	214	72	3.3	14	1319	825	3.28
	RR17	(0)	4	254	93	4.4	10	1548	950	3.54
2^{16}	This	4	12	735	590	36.1	14	7114	6604	35.7
		2	12	811	663	42.4	14	8192	7682	41.82
		1	12	905	762	49.5	14	9425	8930	48.79
		0.5	10	959	817	56.4	12	10620	10109	55.25
	RR17	(0)	4	1318	1005	69.1	10	10651	9932	54.04
2^{20}	This	4	12	10018	9855	619.0	12	109900	109375	619.08
		2	12	10735	10574	728.8	14	128206	127692	720.54
		1	12	12995	12833	853.8	14	150451	149932	846.53
		0.5	10	13560	13398	971.2	12	167438	166914	957.38
	RR17	(0)	4	18478	16295	1302.1	10	191709	189792	1071.65

Table 5.2: The effect of different differential privacy parameters on our protocol performance, compared to the baseline RR17 protocol which has no leakage.

re-running the simulation in a slightly more complex way. Finding a clear and simple characterization of how privacy degrades as adversarily knowledge increases remains an open question.

5.7 Experiments

To now explore the concrete tradeoff between differentially private leakage and performance (computation and communication) in the Rindal-Rosulek PSI protocol. We will see that our protocol offers a smooth trade-off, with performance improving by as much as 50% for some leakage settings.

The baseline for comparison was the Rindal-Rosulek protocol as-is. We obtained the implementation from the authors and modified it to support our variations on reported bin sizes and number of encodings in the final message, as outlined in Section 5.4. One can think of the unmodified Rindal-Rosulek protocol as the special case of our generalization, with $\epsilon = 0$ (i.e., no leakage at all).

With ran all protocols on a single server similar to that used by [RR17b] with the difference that we only consider single-threaded performance. We stress that the only difference in experiments is the bin size and number of final-round encodings, resulting from our differentially private leakage. The benchmark machine has 256GB of RAM and 36 physical cores at 2.6 GHz where a single core is assigned to each party. Two network settings were considered, 1) the LAN setting where latency is less than a millisecond and throughput is 10Gbps of which about 1% was actually utilized. 2) the WAN setting with a 80ms round trip latency and 40Mbps throughput.

Differential Privacy In Figure 5.2 we report the performance of our protocol for differential privacy parameters of $\epsilon \in \{0.5, 1, 2, 4\}$. However, recall that Alice produces two pieces of differentially-private leakage: the overestimate of the bin loads, and the overestimate of the number of encodings (inner product). Of the total ϵ privacy budget, some must be allocated to each of these two mechanisms. We found it advantageous to allocate the lion's share of the privacy budget to the histogram mechanism (since there are a huge number of common encodings, and differential privacy of the inner-product overestimate only contributes an *additive* number of extra encodings). The row in the table corresponding to privacy budget ϵ corresponds to 0.99ϵ-DP load overestimate and 0.01ϵ-DP inner-product overestimate for Alice. The receiver uses the entire privacy budget for the bin-load overestimate.

Compared to the fully-secure case of no leakage, our protocol can be up to 46% faster while providing a similar reduction in communication. Specifically, for larger set sizes such as 2^{20} and an $\epsilon = 4$, our protocol requires 619MB of communication compared to 1302MB of [RR17b]. This almost directly mirrors the reduction in average reported bin sizes, i.e. the effective bins in our protocol with $\epsilon = 4$ are approximately 43% smaller than that of [RR17b]. For a more conservative $\epsilon = 1$, we obverse that our protocol achieves a middle ground between $\epsilon = 4$ and [RR17b], requiring 846MB of communication (an improvement of 35%). For the most conservative $\epsilon = 0.5$, we require only 971MB, which still represents an improvement of 26%.

The main parameters that effects performance, apart from ϵ, is the expected bin size n/m. This parameter controls the relative number of dummy items that occupy each bin. As this value increases, fewer dummy items are required, reducing the number of calls to the encoding functionality. However, this inversely increases the number of common encoding that are sent at the end of the protocol. As such, these trade-offs must be balanced. Our protocol receives additional benefits from larger expected bin sizes due to differential privacy allowing fewer dummy items per bin. For instance, in the LAN setting where communication has minimal impact, [RR17b] found that $n/m = 4$ minimized total latency, while in our setting the best choice for n/m was between 10 and 16. Despite this increase, the number of common encodings sent in our protocols is decreased due to the differentially private leakage.

We see a similar trend in the WAN setting where communication dominates the computational overhead. Here the differentially private leakage allows even larger bin sizes while not significantly increasing the number of common encodings that are sent. This directly result in the difference in communication overhead that is observed.

Distributional Differential Privacy We also explored allowing DDP leakage, as explained in Section 5.6.3. In this setting, Alice can set a threshold t, and for each bin announce whether it contains t or more items. Then each bin is treated as if it has either $t - 1$ items or the worst-case number of items (i.e., whatever the fully-secure protocol would do). Recall that this mechanism can only be used by Alice, since the protocol's output is side information about the private inputs that interferes with the privacy analysis.

Due to the difficulty in obtaining concrete quantitative bounds on the privacy level, we consider only one privacy setting for the DDP case. Specifically, for $n \in \{2^{16}, 2^{20}\}$ items, and $m = n/10$ bins, we let the sender announce whether each bin has less than $t = 13$ items. Following the analysis in Section 5.6.3, we conservatively estimate that this mechanism provides $(\epsilon = 0.33, \delta = 0.00005)$-DDP.

The performance of the DDP mechanism is reported in Figure 5.3. We observe that this result in decreased communication/running time in all cases except when compared to our standard DP protocol with $\epsilon = 4$. For instance, with $n = 2^{20}$ and a $\epsilon = 0.33$ our DDP protocol requires 830MB of communication compared to 957MB of our DP protocol (with a larger epsilon) and 1072MB for [RR17b]. This further decrease in communication is of particular note for the WAN setting where communication is the main bottleneck. However, even in the LAN setting we observe a considerable decrease in running time.

n	Protocol	ϵ	LAN Setting		WAN Setting		Comm. (MB)
			Total Time (ms)	Online Time (ms)	Total Time (ms)	Online Time (ms)	
2^{16}	This	4	755	590	7808	7294	37.36
		2	769	631	8107	7589	40.87
		1	834	682	8774	8262	44.65
		0.33	932	781	9305	8796	47.64
	RR17	(0)	1318	1005	10651	9932	54.04
2^{20}	This	4	8088	7925	115103	114580	646.61
		2	10074	9918	128277	127754	707.13
		1	10256	10096	133911	132153	775.32
		0.33	10428	10270	138327	137143	829.61
	RR17	(0)	18478	16295	191709	189792	1071.65

Table 5.3: The effect of different distributional differential privacy parameters on our protocol performance, compared to the baseline RR17 protocol which has no leakage. All of our protocol executions have expected bin load of $n/m = 10$ and threshold $t = 13$.

5.8 Future Work

Looking more closely at our protocol, the differentially private leakage is on a **random histogram** — i.e., a random function applied to the private input set. Indeed, the nature of this leakage motivates the choice of distributional DP — rather than arguing that an attacker has limited information about some of the private items, it is enough to say that the attacker's auxiliary information about the items is sufficient limited as to make most of the hashes look random.

An interesting future direction is to extend differential privacy definitions and mechanisms to give better accuracy in such a setting — i.e., where we are trying to sanitize information that is already somewhat "random information" about the private inputs.

The most challenging part of our analysis is obtaining good concrete bounds for the DDP privacy analysis. We obtain estimates using intensive computations, but analytical bounds would be more convenient.

There is also an issue that the DDP mechanism can only be applied to one party's histogram. This stems from the fact that DDP does not automatically compose with auxiliary information — yet, the PSI output itself will become auxiliary information but is not known at the time that DDP parameters are chosen. A better understanding of this composition problem would be helpful, since our DDP mechanism gives significantly better performance.

Parameters: X is Alice's input, Y is Bob's input, where $X, Y \subseteq \{0,1\}^\sigma$. m is the number of bins. The protocol uses instances of $\mathcal{F}_{\mathsf{encode}}$ with input length σ, and output length $\lambda + 2\log(n\mu)$, where λ is the security parameter and μ is an upper bound on the number of items in a bin.
Finally, a load-overestimate function $\widetilde{L}$ and an inner-product overestimate function $\widetilde{I}$.

1. **[Parameters]** Parties agree on a random hash function $h : \{0,1\}^\sigma \to [m]$ using a coin tossing protocol.

2. **[Hashing]**

   (a) For $x \in X$, Alice adds x to bin $\mathcal{B}_X[h(x)]$. She computes load overestimate $(\widetilde{a}_1, \ldots, \widetilde{a}_m) \leftarrow \widetilde{L}_h(X)$ and announces this value to Bob. She adds dummy items to the bins so that each bin i has exactly $\widetilde{a}_i$ items (she aborts in the event that there are already more than $\widetilde{a}_i$ items in bin i). The items in each bin $\mathcal{B}_X[i]$ are randomly permuted.

   (b) For $y \in Y$, Bob adds y to bin $\mathcal{B}_Y[h(y)]$. He computes an overestimate $(\widetilde{b}_1, \ldots, \widetilde{b}_m) \leftarrow \widetilde{L}_h(X)$ and computes $(\widetilde{c}_1, \ldots, \widetilde{c}_m) = (\max\{\widetilde{a}_1, \widetilde{b}_1\}, \ldots, \max\{\widetilde{a}_m, \widetilde{b}_m\})$ and announces it to Alice. He adds dummy items until each bin i contains exactly $\widetilde{c}_i$ items.

3. **[Encoding]** For bin index $d \in [m]$:

   (a) For $p \in [\widetilde{a}_d]$, let x be the pth item $\mathcal{B}_X[d]$. Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, d, p), x)$ to the $\mathcal{F}_{\mathsf{encode}}$ functionality which responds with $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, d, p), [\![x]\!]^{\mathsf{B}}_{d,p})$. Bob receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, d, p))$ from $\mathcal{F}_{\mathsf{encode}}$.

   (b) For $p \in [\widetilde{c}_d]$, let y be the pth item $\mathcal{B}_Y[d]$. Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, d, p), y)$ to the $\mathcal{F}_{\mathsf{encode}}$ functionality which responds with $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, d, p), [\![y]\!]^{\mathsf{A}}_{d,p})$. Alice receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, d, p))$ from $\mathcal{F}_{\mathsf{encode}}$.

4. **[Output]**

   (a) **[Alice's Common Mask]** For each $x \in X$, in random order, let d, p be the bin index and position that x was placed in during Step 2a. For $j \in [\widetilde{c}_d]$, Alice sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{A}, d, j), x)$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{A}, d, j), [\![x]\!]^{\mathsf{A}}_{d,j})$ in response. Alice adds $[\![x]\!]^{\mathsf{A}}_{d,j} \oplus [\![x]\!]^{\mathsf{B}}_{d,p}$ to a set E.

   (b) Now E contains $\mathsf{Ld}_h(X) \cdot (\widetilde{c}_1, \ldots, \widetilde{c}_m)$ items. Alice computes inner-product overestimate $\widetilde{e} \leftarrow \widetilde{I}(\mathsf{Ld}_h(X), (\widetilde{c}_1, \ldots, \widetilde{c}_m))$ and adds randomly chosen values to E until $|E| = \widetilde{e}$. She sends E (randomly permuted) to Bob.

   (c) **[Bob's Common Mask]** Similarly, for $y \in Y$, let d, p be the bin index and position that y was placed in during Step 2b. For $j \in [\widetilde{a}_d]$, Bob sends $(\textsc{Encode}, (\mathsf{sid}, \mathsf{B}, d, j), y)$ to $\mathcal{F}_{\mathsf{encode}}$ and receives $(\textsc{Output}, (\mathsf{sid}, \mathsf{B}, d, j), [\![y]\!]^{\mathsf{B}}_{d,j})$ in response. Bob outputs

$$\left\{ y \in Y \;\middle|\; \exists j : [\![y]\!]^{\mathsf{A}}_{d,p} \oplus [\![y]\!]^{\mathsf{B}}_{d,j} \in E \right\}$$

Figure 5.3: One-sided [illegible] PSI protocol (with leakage)

Chapter 6

PSI From Fully Homomorphic Encryption

Fast Private Set Intersection from Homomorphic Encryption by Hao Chen, Kim Laine & Peter Rindal, in CCS [CLR17]

6.1 Introduction

Over the last few years, PSI has become truly practical for a variety of applications due to a long list of publications, e.g. [PSZ14, PSSZ15, PSZ18, KKRT16, OOS17, RR16, Lam16, BFT16, DCW13]. The most efficient *semi-honest* protocols have been proposed by Pinkas *et al.* [PSZ18] and Kolesnikov *et al.* [KKRT16]. While these protocols are extremely fast, their communication complexity is linear in the sizes of both sets. When one set is significantly smaller than the other, the communication overhead becomes considerable compared to the non-private solution, which has communication linear in the size of the smaller set.

6.1.1 Chapter Contributions

As our discussion has shown, all of the prior PSI protocols require both parties to encode and send data over the network that is proportional to their entire sets. However, the trivial insecure solution only requires the smaller set to be sent. We address this gap by constructing the first secure and practical PSI protocol with low communication overhead based on a leveled fully homomorphic encryption scheme.

Our basic protocol requires communication linear in the smaller set, achieving optimal communication that is on par with the naive solution. We then combine an array of optimizations to significantly reduce communication size, computational cost, and the depth of the homomorphic circuit, while only adding a logarithmic overhead to the communication. In summary, we

- Propose a basic PSI protocol based on fully homomorphic encryption;
- Combine various optimizations to vastly reduce the computational and communication cost;
- Use fine-tuned fully homomorphic encryption parameters for the homomorphic computation to avoid the costly *bootstrapping* operation [Gen09, GHS12b], and to achieve good performance;
- Develop a prototype implementation in C++ and demonstrate a 38–115× reduction in communication over previous state-of-the-art protocols.

In Section 6.1.2 we review the setups and tools we use to build the protocol: the PSI setup and its definition of security, and preliminaries on (leveled) fully homomorphic encryption. In Section 6.2 we propose our basic strawman PSI protocol. Then, in Section 6.3, we apply optimizations to vastly improve the strawman protocol and make it practical. The formal description of the optimized protocol, along with a security proof, is presented in Section 6.4. In Section 6.5 we provide a performance analysis of our implementation, and compare our performance results to [PSZ18] and [KKRT16].

6.1.2 Notations

Throughout this chapter, we will use the notation:

- $X, Y \subseteq \{0,1\}^\sigma$ are Alice's and Bob's sets, each of size N_{X}, N_{Y};
- m denotes the size of a hash table, and d denotes the number of items to be inserted into a hash table;
- n, q and t denote the encryption parameters described in Section 2.1.4;
- h denotes the number of hash functions used for cuckoo hashing in Section 6.3.2;

- B denotes the bin size for the simple hashing scheme described in Section 6.3.2;
- ℓ denotes the windowing parameter described in Section 6.3.3;
- α denotes the partitioning parameter described in Section 6.3.3.

6.1.3 Unbalanced Private Set Intersection

Our protocol is particularly powerful when Alice's set is much larger than Bob's set. Hence we assume $N_{\mathrm{X}} \gg N_{\mathrm{Y}}$ throughout the paper, even though the protocol works for arbitrary set sizes with no changes. More precisely, we achieve a communication complexity of $O(N_{\mathrm{Y}} \log N_{\mathrm{X}})$. Also, we require only Alice to perform work linear in the larger set size N_{X}. Intuitively, Bob encrypts and sends its set to Alice, who computes the intersection on homomorphically encrypted data by evaluating an appropriate comparison circuit. The output is then compressed to much smaller size using homomorphic multiplication, and sent back to Bob for decryption. We note that Bob only performs relatively light computation in the protocol, i.e. encryptions and decryptions of data linear in its set size N_{Y}. This is particularly useful when Bob is limited in its computational power, e.g. when Bob is a mobile device.

Private contact discovery

One particularly interesting application for our PSI protocol is *private contact discovery* which review in detail now. In this setting, a service provider, e.g. *WhatsApp*, has a set of several million users. Each of these users holds their own set of contacts, and wants to learn which of them also use the service. The insecure solution to this problem is to have the user send the service provider their set of contacts, who then performs the intersection on behalf of the user. While this protects the privacy of the service provider, it leaks the user's private contacts to the service provider.

While PSI offers a natural solution to this problem, one potential issue with applying existing protocols to this setting is that both the communication and computation complexity for both parties is linear in the larger set. As a result, a user who may have only a few hundred contacts has to receive and process data linear in the number of users that the service has, resulting in a suboptimal protocol for constrained hardware, such as cellphones. This problem was initially raised in an article by Moxie Marlinspike from *Open Whisper Systems*—the company that developed the popular secure messaging app

Signal—when they were trying to deploy PSI for contact discovery [Mar14]. Our PSI protocol addresses this issue by allowing the constrained devices to process and receive data that is linear in only their set size, and only logarithmic in the service provider's set size. Moreover, the major part of the computation can be performed by the service provider in a large data center, where processing power is relatively inexpensive, whereas the user only performs a light computation.

6.2 The Basic Protocol

We describe our basic protocol in Figure 6.1 as a strawman protocol. Bob encrypts each of its items y, and sends them to Alice. For each y, Alice then evaluates homomorphically the product of differences of y with all of Alice's items x, randomizes the product by multiplying it with a uniformly random non-zero plaintext, and sends the result back to Bob. The result decrypts to zero precisely when y is in Alice's set, and to a uniformly random non-zero plaintext otherwise, revealing no information about Alice's set to Bob.

To be more precise, we assume from now on that the plaintext modulus t in our FHE scheme is a prime number, large enough to encode σ-bit strings as elements of $\mathbb{Z}_t$. We also temporarily restrict the plaintext space to its subring of constant polynomials (this restriction will be removed in Section 6.3.1), and assume plaintexts to be simply elements of $\mathbb{Z}_t$. Recall that the sizes of the sets X and Y, and the (common) bit-length σ of the items, are public information.

We have the following informal theorem with regards to the security and correctness of the basic protocol.

Theorem 17 (informal)**.** *The protocol described in Figure 6.1 securely and correctly computes the private set intersection of X and Y in the semi-honest security model, provided that the fully homomorphic encryption scheme is IND-CPA secure and achieves circuit privacy.*

Proof sketch. Receiver's security is straightforward: Bob sends an array of ciphertexts, which looks pseudorandom to Alice since the fully homomorphic encryption scheme is IND-CPA secure. For sender's security, we note that Bob's view consists of an array of ciphertexts. It follows from circuit privacy that Bob only learns the decryptions of these ciphertexts, and nothing more.

Input: Receiver inputs set Y of size N_{Y}; sender inputs set X of size N_{X}. Both sets consist of bit strings of length σ. N_{X}, N_{Y}, and σ are public.
Output: Receiver outputs $X \cap Y$; sender outputs $\perp$.

1. **Setup**: Sender and receiver jointly agree on a fully homomorphic encryption scheme. Receiver generates a public-secret key pair for the scheme, and keeps the secret key to itself.

2. **Set encryption**: Receiver encrypts each element y_i in its set Y using the fully homomorphic encryption scheme, and sends the N_{Y} ciphertexts $(c_1, \ldots, c_{N_{\mathrm{Y}}})$ to sender.

3. **Computing intersection**: For each c_i, sender

   (a) samples a random non-zero plaintext element r_i;

   (b) homomorphically computes

   $$d_i = r_i \prod_{x \in X} (c_i - x) \, .$$

   Sender return the ciphertexts $(d_1, \ldots, d_{N_{\mathrm{Y}}})$ to receiver.

4. **Reply extraction**: Receiver decrypts the ciphertexts $(d_1, \ldots, d_{N_{\mathrm{Y}}})$ and outputs

   $$X \cap Y = \{ y_i : \mathsf{FHE.Decrypt}(d_i) = 0 \} \, .$$

Figure 6.1: Basic FHE based PSI protocol.

For a fixed index i, we have

$$\mathsf{FHE.Decrypt}(d_i) = r_i \prod_{x \in X} (y_i - x) \, ,$$

which is zero precisely when $y_i \in X$ (correctness), and otherwise a uniformly random element in $\mathbb{Z}_t \setminus \{0\}$, because $\mathbb{Z}_t$ is a field. Thus, Bob learns no additional information beyond the intersection $X \cap Y$. □

This basic strawman protocol is extremely inefficient: it requires Alice to perform $O(N_{\mathrm{X}} N_{\mathrm{Y}})$ homomorphic multiplications and additions, and the depth of the circuit is high, pushing the FHE parameter sizes to be huge. In addition, Alice and Bob need to communicate $O(N_{\mathrm{Y}})$ FHE ciphertexts, which can be prohibitive even for state-of-the-art fully homomorphic encryption schemes. It is therefore quite surprising that the protocol becomes very efficient when combined with the enhancements described in the next section.

6.3 Optimizations

6.3.1 Batching

Our first step to improve performance is through the use of *batching*, which is a well-known and powerful technique in fully homomorphic encryption to enable SIMD (Single Instruction, Multiple Data) operations on ciphertexts. We give a brief explanation here, and refer the reader to [GHS12b, BGH13, SV14, KL16, GBDL+16] for more details and example applications.

For suitable choices of the plaintext modulus t, there is a ring isomorphism from the plaintext space R_t to $\mathbb{Z}_t^n$. As an example, a constant polynomial $a \in R_t$ corresponds to the vector $(a, \ldots, a) \in \mathbb{Z}_t^n$. Moreover, this isomorphism translates polynomial additions and multiplications into coefficient-wise additions and multiplications in each of the n fields $\mathbb{Z}_t$. To simplify the exposition, we use the polynomial and vector notations for plaintexts interchangeably, omitting the conversions from one representation to the other.

We can apply batching to reduce both the computational and communication cost of the basic protocol as follows. Bob groups its items into vectors of length n, encrypts them, and sends N_{Y}/n ciphertexts to Alice. Upon seeing each ciphertext c_i, Alice samples a *vector* $r_i = (r_{i1}, \ldots, r_{in}) \in (\mathbb{Z}_t^*)^n$ of uniformly random non-zero elements of $\mathbb{Z}_t$, homomorphically computes $d_i = r_i \prod_{x \in X}(c_i - x)$, and sends it back to Bob. Note that these modifications do not affect correctness or security, since the exact same proof can be applied per each vector coefficient.

The batching technique allows Alice to operate on n items from Bob simultaneously, resulting in n-fold improvement in both the computation and communication. Since in typical cases n has size several thousands, this results in a significant improvement over the basic protocol.

6.3.2 Hashing

Even with the batching techniques of Section 6.3.1, Alice still needs to encode each of its set elements into separate plaintexts, and individually compare them to Bob's items. Instead, it would be nice if Alice could also take advantage of batching. We will achieve this through the use of hashing techniques. Specifically, we use batching in conjunction with *cuckoo hashing* and *permutation-based hashing*, which have been developed and explored in detail in the context of PSI in e.g. [PSZ14, PSSZ15].

Before jumping into the technicalities of cuckoo hashing and permutation-based hashing, we start with a high-level explanation of why hashing is beneficial in our context. Suppose the two parties hash the items in their sets into two hash tables using some agreed-upon deterministic hash function. Now they only need to perform a PSI for each bin, since items in different bins are necessarily different.

One important point is that all bins must be padded to a fixed size to maintain security. Observe that the bins prior to padding will have uneven loads, and the load of a specific bin (the number of items mapped into the bin) can reveal additional information beyond the intersection. To overcome this, we need to pad each bin with dummy items up to a pre-determined maximum size.

The *simple hashing* technique just described significantly reduces the complexity of our protocol. It is well known that hashing d items into a hash table of size $m = d$ results in a maximum load of $O(\log d)$ with high probability. For example, in the case that both parties have $d = N_{\text{X}} = N_{\text{Y}}$ items, the overall complexity of the basic protocol reduces to $O(d \log^2 d)$, where the $\log^2 d$ factor comes from performing the basic PSI protocol on a single bin. Next, we will reduce the complexity even further via better hashing techniques.

Cuckoo hashing

Cuckoo hashing [PR01, DM03, FPSS03] is a way to build dense hash tables by using $h > 1$ hash functions $H_1, ..., H_h$. To insert an item x, we choose a random index i from $[h]$, and insert the tuple (x, i) at location $H_i(x)$ in the table. If this location was already occupied by a tuple (y, j), we replace (y, j) with (x, i), choose a random j' from $[h] \setminus \{j\}$, and recursively re-insert (y, j') into the table. For $m \approx d$ and fairly small h, cuckoo hashing succeeds with very high probability, i.e. the recursive re-insertion process always succeeds before a pre-determined upper bound on the recursion depth is reached. We will discuss the success probability of cuckoo hashing in Section 6.3.2.

In order to apply cuckoo hashing to our PSI protocol, we must ensure that bin-wise comparisons will always yield the correct intersection. This is done by letting Bob perform cuckoo hashing with $m \gtrsim N_{\text{Y}}$ bins. Alice must insert each of its items into a two-dimensional hash table using *all* h hash functions $H_1, ..., H_h$ (simple hashing), because there is no way for it to know which one of the hash functions Bob eventually ended up using for the items in the intersection. To determine the maximum load on Alice's side, we apply a standard balls-into-bins argument. Concretely, when inserting $d = hN_{\text{x}}$ balls into m bins, we have

$$\begin{aligned} &\Pr[\text{at least one bin has load } > B] \\ &\leq m \sum_{i=B+1}^{d} \binom{d}{i} \left(\frac{1}{m}\right)^{i} \left(1 - \frac{1}{m}\right)^{d-i} . \end{aligned} \tag{6.1}$$

Our default assumption is that Alice (who performs simple hashing) has a larger set, so that $d > m \log m$. In this case B is upper-bounded by $d/m + O(\sqrt{d \log m/m})$ with high probability [RS98].

Permutation-based hashing

Independent of the exact hashing scheme, permutation-based hashing [ANS10b] is an optimization to reduce the length of the items stored in the hash tables by encoding a part of an item into the bin index. For simplicity, we assume m is a power of two, and describe permutation-based hashing only in connection with cuckoo hashing. To insert a bit string x into the hash table, we first parse it as $x_L \| x_R$, where the length of x_R is equal to $\log_2 m$. The hash functions $H_1, ..., H_h$ are used to construct location functions as

$$\texttt{Loc}_i(x) = H_i(x_L) \oplus x_R \,, \quad 1 \leq i \leq h \,,$$

which we will use in cuckoo hashing. Moreover, instead of inserting the entire tuple (x, i) into the hash table as in regular cuckoo hashing, we only insert (x_L, i) at the location specified by $\texttt{Loc}_i(x)$.

The correctness of the PSI protocol still holds after applying permutation-based hashing. The reason is if $(x_L, i) = (y_L, j)$ for two items x and y, then $i = j$ and $x_L = y_L$. If in addition these are found in the same location, then $H_i(x_L) \oplus x_R = H_j(y_L) \oplus x_R = H_j(y_L) \oplus y_R$, so $x_R = y_R$, and hence $x = y$. The lengths of the strings stored in the hash table are thus reduced by $\log_2 m - \lceil \log_2 h \rceil$ bits. The complete hashing routine is specified in Figure 6.2.

Input: Receiver inputs set Y of size N_{Y}; sender inputs set X of size N_{X}. Both sets consist of bit strings of length σ. N_{X}, N_{Y}, σ are public. Both parties input integers h, m, B and a set of hash function $H_1, ..., H_h : \{0,1\}^{\sigma - \log_2 m} \to \{0,1\}^{\log_2 m}$. The location functions $\mathtt{Loc}_i$ is defined with respect to H_i for $i \in [h]$.
Output: Receiver outputs a permutation-based cuckoo hash table with the items in Y inserted, or $\perp$. Sender outputs a permutation-based hash table with the items in X inserted using simple hashing and all location functions, or $\perp$.

1. **[Sender]** Let $\mathfrak{B}_{\text{X}}$ be an array of m bins, each with capacity B, and value $\{(\perp, \perp)\}^B$. For each $x \in X$ and $i \in [h]$, Alice samples $j \leftarrow [B]$ s.t. $\mathfrak{B}_{\text{X}}[\mathtt{Loc}_i(x)][j] = \perp$, and sets $B_{\text{X}}[\mathtt{Loc}_i(x)][j] := (x_L, i)$. If the sampling fails due to a bin being full, Alice outputs $\perp$. Otherwise it outputs $\mathfrak{B}_{\text{X}}$.

2. **[Receiver]** Let $\mathfrak{B}_{\text{Y}}$ be an array of m bins, each with capacity 1, and value $(\perp, \perp)$. For each $y \in Y$, Bob

   (a) sets $w = y$, and $i \leftarrow [B]$;

   (b) defines and calls the function $\mathsf{Insert}(w, i)$ as follows: swap (w, i) with the entry at $\mathfrak{B}_{\text{Y}}[\mathtt{Loc}_i(w)]$. If $(w, i) \neq (\perp, \perp)$, recursively call $\mathsf{Insert}(w, j)$, where $j \leftarrow [h] \setminus \{i\}$.

   If for any $y \in Y$ the recursive calls to Insert exceeds the system limit, Bob halts and outputs $\perp$. Otherwise it outputs $\mathfrak{B}_{\text{Y}}$.

Figure 6.2: Hashing routine for the FHE based PSI protocol.

Hashing failures

In an unlikely event where cuckoo hashing fails, it could leak some information of Bob's set to Alice. To prevent this, we must ensure that with overwhelming probability the cuckoo hashing algorithm will succeed. While some asymptotic results exist for estimating the failure probability of cuckoo hashing [FMM09, DGM+10], the hidden constants are difficult to determine precisely. Instead, to obtain optimal parameters, we choose to determine the failure probability using empirical methods. The general technique we use is similar to that of [PSZ18], with two exceptions: first, we omit an auxiliary data structure known as the *stash* due to its incompatibility with the fully homomorphic encryption approach; second, we primarily focus on $h = 3$ in our experiments (see below), whereas [PSZ18] focused on $h = 2$.

We start by fixing the cuckoo hash table consisting of m bins, and vary the

Table size m	Insert size d											
	$3 \cdot 2^{8}$		$3 \cdot 2^{12}$		$3 \cdot 2^{16}$		$3 \cdot 2^{20}$		$3 \cdot 2^{24}$		$3 \cdot 2^{28}$	
	$\lambda = 30$	40	30	40	30	40	30	40	30	40	30	40
8192	8	9	17	20	68	74	536	556	6727	6798	100611	100890
16384	7	8	13	16	46	51	304	318	3492	3543	50807	51002

Table 6.1: Simple hashing bin size upper bound B for failure probability $2^{-\lambda}$, with $\lambda \in \{30, 40\}$, and $h = 3$; see equation (6.1).

number for items $d < m$ to be inserted. For each (d, m) pair, we run the cuckoo hashing algorithm 2^{30} times. For $d \ll m$, we find that the algorithm never fails in the experiments. To compute the required ratio $\epsilon = m/d$ to achieve a statistical security level of $\lambda \geq 40$ (i.e. cuckoo hashing fails with probability at most 2^{-40}), we begin by setting ϵ to a value slightly larger than one, and gradually increase it until we can expect zero hashing failures. From this we observe that λ increases linearly with the scaling factor ϵ when $h \geq 3$.

Over the course of our experiments, we observed that cuckoo hashing with no stash performs very poorly when $h = 2$, which was also observed and discussed in detail in [PSZ18], which is why we shift our focus to $h = 3$. Furthermore, the marginal gain of $h = 4$ is outweighed by the increased cost of simple hashing. By applying linear regression to the empirical data for $\lambda \geq 0$, we observe that $\lambda = 124.4\epsilon - 144.6$ for $m = 16384$, and $\lambda = 125\epsilon - 145$ for $m = 8192$. To achieve a statistical security level of $\lambda = 40$, the maximum number of items that can be cuckoo hashed into 8192 bins with $h = 3$ is therefore 5535. For $m = 16384$, the corresponding maximum number of items is 11041. The respective simple hashing parameter for the given hash table size and different $d = hN_{\mathrm{X}}$ values are given in Table 6.1.

Dummy values

In order to make Alice's simple hash table evenly filled, we need to pad each bin with dummy items after hashing. We let Alice and receiver fix two different dummy values from $\mathbb{Z}_t$, as long as they do not occur as legitimate values. For example, if legitimate values have at most σ bits, then we can set Bob's dummy value to 2^{σ}, and Alice's dummy value to $2^{\sigma+1} - 1$.

Hashing to a smaller representation

In many cases the total number of items $N_{\mathrm{X}} + N_{\mathrm{Y}}$ is much smaller than the number 2^{σ} of all possible strings of length σ. Since the performance of our protocol will degrade with increasing string length, it is beneficial for the parties to compress their strings with an agreed-upon hash function to a fixed length $\sigma_{\max}$, and then execute the PSI protocol on these hashed strings. Indeed, this is a well-known technique in the PSI community.

More precisely, when a total of $N_{\mathrm{X}} + N_{\mathrm{Y}}$ random strings are hashed to a domain of size $2^{\sigma_{\max}}$, the probability of a collision is approximately $(N_{\mathrm{X}} + N_{\mathrm{Y}})^2/2^{\sigma_{\max}+1}$. For a statistical security parameter λ, we require that $\Pr[\text{collision occurs}] \leq 2^{-\lambda}$. Therefore, the compressed strings should have length at least

$$\sigma_{\max} = 2\log_2(N_{\mathrm{X}} + N_{\mathrm{Y}}) + \lambda - 1.$$

Now we apply permutation-based cuckoo hashing to the compressed strings, further reducing the string length to

$$\sigma_{max} - \log_2 m + \lceil \log_2 h \rceil.$$

In addition, we need to reserve two more values in the plaintext space for the dummy values discussed in Section 6.3.2. Thus, by choosing the encryption parameter t so that

$$\log_2 t > \sigma_{max} - \log_2 m + \lceil \log_2 h \rceil + 1 \tag{6.2}$$

we can accommodate arbitrarily long strings in our PSI protocol.

Combining with batching

It is straightforward to combine hashing techniques introduced in this section with the batching technique in Section 6.3.1. After Bob hashes its items into a table of size m, it parses the table into m/n vectors of length n. Bob then encrypts each vector using batching, and proceeds as usual. Similarly, Alice performs the same batching step for each of the B columns of its two-dimensional hash table, resulting in Bm/n plaintext vectors. The rest of the protocol remains unchanged, and we see that adding batching to the hashing techniques provides an n-fold reduction in both computation and communication.

6.3.3 Reducing the Circuit Depth

With the optimizations discussed in Section 6.3.1 and Section 6.3.2, our protocol already achieves very low communication cost: typically just a few homomorphically encrypted ciphertexts. Unfortunately, the depth of the arithmetic circuit that needs to be homomorphically evaluated is still $O(\log N_{\mathrm{X}})$, which can be prohibitively high for currently known fully homomorphic encryption schemes.

We use two tricks—*windowing* and *partitioning*—to critically reduce this depth. For simplicity of exposition, we will discuss how these two tricks work over the basic protocol, and briefly explain how to combine them with previous optimizations.

Windowing

We use a standard windowing technique to lower the depth of the arithmetic circuit that Alice needs to evaluate on Bob's homomorphically encrypted data, resulting in a valuable computation-communication trade-off.

Recall that in the basic protocol, for each item $y \in Y$, Bob sends one ciphertext $c = \mathsf{FHE.Encrypt}(y)$ to Alice, who samples a random element r in $\mathbb{Z}_t \setminus \{0\}$, homomorphically evaluates $r \prod_{x \in X}(c - x)$, and sends the result back to Bob. If Bob sends encryptions of extra powers of y, Alice can use these powers to evaluate the same computation with a much lower depth circuit. More precisely, for a *window size* of ℓ bits, Bob computes and sends $c^{(i,j)} = \mathsf{FHE.Encrypt}(y^{i \cdot 2^{\ell j}})$ to Alice for all $1 \leq i \leq 2^{\ell} - 1$, and all $0 \leq j \leq \lfloor \log_2(N_{\mathrm{X}})/\ell \rfloor$. For example, when $\ell = 1$, Bob sends encryptions of $y, y^2, y^4, \ldots, y^{2^{\lfloor \log_2 N_{\mathrm{X}} \rfloor}}$.

This technique results in a significant reduction in the circuit depth. To see this, we write

$$r \prod_{x \in X}(y - x) = r y^{N_{\mathrm{X}}} + r a_{N_{\mathrm{X}}-1} y^{N_{\mathrm{X}}-1} + \ldots + r a_0 \,. \tag{6.3}$$

If Alice only has an encryption of y, it needs to compute *at worst* the product $r y^{N_{\mathrm{X}}}$, which requires a circuit of depth $\lceil \log_2(N_{\mathrm{X}}+1) \rceil$. Now if the encryptions $c^{(i,j)}$ are already given to Alice, then we can separate Alice's computation into two steps. First, Alice computes an encryption of y^i for all $0 \leq i \leq N_{\mathrm{X}}$. Alice needs to compute *at worst* a product of $\lfloor \log_2(N_{\mathrm{X}})/\ell \rfloor + 1$ terms, requiring a circuit of depth $\lceil \log_2(\lfloor \log_2(N_{\mathrm{x}})/\ell \rfloor + 1) \rceil$. In an extreme case, if Bob gives Alice encryptions of all powers of y up to $y^{N_{\mathrm{x}}}$, the depth in this step becomes zero. Then, Alice computes a dot product of encryptions of y^i $(0 \leq i \leq N_x)$ with the vector of coefficients $(r, ra_{N_{\mathrm{x}}-1}, \dots, ra_0)$ in plaintext from its own data. This second step has multiplicative depth one.

The cost of windowing is in increased communication. The communication from Bob to Alice is increased by a factor of $(2^\ell - 1)(\lfloor \log_2(N_{\mathrm{x}})/\ell \rfloor + 1)$, and the communication back from Alice to Bob does not change.

It is easy to incorporate batching and hashing methods with windowing. The only difference is that batching and hashing effectively reduce Alice's set size by nearly a factor of n. More precisely, the depth of the circuit becomes $\lceil \log_2(\lfloor \log_2(B)/\ell \rfloor + 1) \rceil + 1$, where B is as in Figure 6.2. Without windowing, batching and hashing encode the entire set Y into one hash table of size $m \gtrsim N_Y$, producing m/n ciphertexts to be communicated to Alice. With windowing this is expanded to $(2^\ell - 1)(\lfloor \log_2(B)/\ell \rfloor + 1) \cdot m/n$ ciphertexts.

Finally, we note that security of windowing technique is guaranteed by the IND-CPA security of the underlying fully homomorphic encryption scheme.

Partitioning

Another way to reduce circuit depth is to let Alice partition its set into α subsets, and perform one PSI protocol execution per each subset. In the basic protocol, this reduces sender's circuit depth from $\lceil \log_2(N_{\mathrm{x}} + 1) \rceil$ to $\lceil \log_2(N_{\mathrm{x}}/\alpha + 1) \rceil$, at the cost of increasing the return communication from sender to receiver by a factor of α.

Partitioning can be naturally combined with windowing in a way that offers an additional benefit of reducing the number of homomorphic operations. Recall from Section 6.3.3 that Alice needs to compute encryptions of all powers $y, \dots, y^{N_{\mathrm{x}}}$ for each of Bob's items y. With partitioning, Alice only needs to compute encryptions of $y, \dots, y^{N_{\mathrm{x}}/\alpha}$, which it can reuse for each of the α partitions. Thus, with both partitioning and windowing, Alice's computational cost in the first step described in Section 6.3.3 reduces by a factor of α, whereas the cost in the second step remains the same.

We may combine batching and hashing with partitioning in the following way. Alice performs its part of the hashing routine (Figure 6.2) as usual, but splits the *contents* of its bins (each of size B) into α parts of equal size, resulting in α tables each with bin size $\approx B/\alpha$. It then performs the PSI protocol with the improvements described in Section 6.3.1, 6.3.2, and 6.3.3 using each of the α hash tables. Now sender's circuit depth reduces to $\lceil \log_2(\lfloor \log_2(B/\alpha)/\ell \rfloor + 1) \rceil + 1$, where B is as in Figure 6.2. The communication from Alice to Bob is α ciphertexts.

We would like to note that in order to preserve Alice's security, it is essential that after using simple hashing to insert its items into the hash table, Alice partitions the contents of the bins—including empty locations with value $(\bot, \bot)$—in a uniformly random way. Since in the hashing routine (Figure 6.2) Alice inserts its items in random locations within each bin, the correct partitioning can be achieved by evenly splitting the contents of each bin into α subsets using any deterministic partitioning method.

6.3.4 Reducing Reply Size via Modulus Switching

Finally, we employ *modulus switching* (see [BGV12]), which effectively reduces the size of the response ciphertexts. Modulus switching is a well-known operation in lattice-based fully homomorphic encryption schemes. It is a public operation, which transforms a ciphertext with encryption parameter q into a ciphertext encrypting the same plaintext, but with a smaller parameter $q' < q$. As long as q' is not too small, correctness of the encryption scheme is preserved. Since FHE ciphertexts have size linear in $\log q$, modulus switching reduces ciphertext sizes by a factor of $\log q / \log q'$. This trick allows Alice to "compress" the return ciphertexts before sending them to Bob. In practice, we are able to reduce the return ciphertexts to about 15–20% of their original size. We note that the security of the protocol is trivially preserved as long as the smaller modulus q' is determined at setup.

6.4 Full Protocol and Security Proof

6.4.1 Formal Description

We detail the full protocol in Figure 6.4, given a secure fully homomorphic encryption scheme with circuit privacy. The ideal functionality of this protocol is given in Figure 6.3.

Parameters: Two parties denoted as Alice and receiver with sets of items of bit-length σ. Receiver's set is of size N_{Y}; sender's set is of size N_{X}. sid denotes the session ID of the protocol instance.

Functionality: On input $(\textsc{Receive}, \mathsf{sid}, Y)$ from Bob and $(\textsc{Send}, \mathsf{sid}, X)$ from Alice, where $X, Y \subseteq \{0,1\}^{\sigma}$, $|X| = N_{\mathrm{X}}$, $|Y| = N_{\mathrm{Y}}$. The functionality sends $(\textsc{Output}, \mathsf{sid}, X \cap Y)$ to Bob, and nothing to Alice.

Figure 6.3: Ideal functionality $\mathcal{F}_{\mathrm{PSI}}$ for private set intersection with one-sided output.

We prove security in the standard *semi-honest* simulation-based paradigm. Loosely put, we say that the protocol Π_{PSI} of Figure 6.4 securely realizes the functionality $\mathcal{F}_{\mathrm{PSI}}$, if it is correct, and there exist two simulators (PPT algorithms) $\mathsf{Sim}_{\mathrm{R}}, \mathsf{Sim}_{\mathrm{S}}$ with the following properties. The simulator $\mathsf{Sim}_{\mathrm{R}}$ takes Bob's set and the intersection as input, and needs to generate a transcript for the protocol execution that is indistinguishable from Bob's view of the real interaction. $\mathsf{Sim}_{\mathrm{S}}$ is similarly defined, with the exception of not taking the intersection as input. For a formal definition of simulation based security in the semi-honest setting, we refer the reader to [Lin16].

Theorem 18. *The protocol in Figure 6.4 is a secure protocol for $\mathcal{F}_{\mathrm{PSI}}$ in the semi-honest setting.*

Proof. It is easy to see that the protocol correctly computes the intersection conditioned on the hashing routine succeeding, which happens with overwhelming probability $1 - 2^{-\lambda}$.

We start with a corrupt receiver, and show the existence of $\mathsf{Sim}_{\mathrm{R}}$. For easy of exposition, we will assume that the simulator/protocol is parameterized by $(h, m, B, n, q, t, \alpha, \ell, H', \{H_i\}_{1 \le i \le h})$, which are fixed and public, and that hashing to a smaller representation (Section 6.3.2) is used. We will then define Bob's simulator $\mathsf{Sim}_{\mathrm{R}}$ as follows. $\mathsf{Sim}_{\mathrm{R}}$ computes the set $Y' = H'(Y)$, and uses a modified hashing routine to cuckoo-hash its elements into a table of size m. The modification is that if an element y is in $X \cap Y$, then a 0 is inserted, and otherwise a random non-zero element in $\mathbb{Z}_t$ is inserted. After hashing finishes, $\mathsf{Sim}_{\mathrm{R}}$ inserts random non-zero elements from $\mathbb{Z}_t$ into the remaining empty slots. Next, $\mathsf{Sim}_{\mathrm{R}}$ creates $\alpha - 1$ more tables of the same size, and fills them with random non-zero elements from $\mathbb{Z}_t$. It then randomly permutes the values inserted in the matching bins among all α tables. Finally, it batches each table into m/n FHE plaintext polynomials, and homomorphically encrypts them into m/n ciphertexts. The resulting

$m/n \cdot \alpha$ ciphertexts will serve as a simulation of Bob's view. Due to the circuit privacy assumption on underlying fully homomorphic encryption scheme, this view is indistinguishable from Bob's view in the real execution of the protocol.

The case of a corrupt sender is straightforward. The simulator $\mathsf{Sim}_{\mathrm{S}}$ can generate new encryptions of zero in place of the encryptions in Step 5. By the IND-CPA security of the fully homomorphic encryption scheme, this result is indistinguishable from Alice's view in the real protocol.

□

6.4.2 Discussion

Function privacy

While our protocol (Figure 6.4) assumes a fully homomorphic encryption scheme with circuit privacy, in practice it is much more efficient to instantiate it with leveled fully homomorphic encryption (recall Section 2.1.4), i.e. choose encryption parameters large enough to avoid the costly bootstrapping operation. This does not change the security properties of the protocol, as the encryption parameters are selected purely based on public parameters N_{X}, N_{Y} and σ.

While circuit privacy can be achieved in fully homomorphic encryption using e.g. the techniques of [DS16], in practice the slightly weaker notion of (statistical) *function privacy* [GHV10] suffices, and is easier to achieve in the leveled setting using *re-randomization* and *noise flooding*, where Alice re-randomizes the output ciphertexts by homomorphically adding to them an encryption of zero with a very large noise [Gen09, DS16]. A standard "smudging lemma" (see e.g. [AJLA+12]) implies that in order to achieve $2^{-\lambda}$ statistical distance between output ciphertexts of different executions, it suffices to add encryptions of zero with noise $\lambda + \log_2 n + \log_2 \alpha$ bits larger than an upper bound on the noise in the original outputs of the computation. We used the heuristic results in [CS16] to bound the amount of noise in the output ciphertexts before flooding.

Input: Receiver inputs set $Y \subset \{0,1\}^\sigma$ of size N_Y; sender inputs set $X \subset \{0,1\}^\sigma$ of size N_X. $N_\text{X}, N_\text{Y}, \sigma$ are public. κ and λ denote the computational and statistical security parameters, respectively.
Output: Bob outputs $Y \cap X$; Alice outputs $\perp$.

1. **[Perform hashing]** Hashing parameters h, m, B are agreed upon such that simple hashing hN_X balls into m bins with max load B, and cuckoo hashing N_Y balls into m bins succeed with probability $\geq 1 - 2^{-\lambda}$.

    (a) **[Hashing to shorter strings]** Let $\sigma' = 2\log_2(N_\text{X} + N_\text{Y}) + \lambda - 1$. If $\sigma > \sigma'$, then both parties hash their sets to a smaller representation. First, a random hash function $H' : \{0,1\}^\sigma \to \{0,1\}^{\sigma'}$ is sampled. Let $X' = \{H'(x) \mid x \in X\}$ and $Y' = \{H'(y) \mid y \in Y\}$. Perform the rest of the protocol with (X', Y', σ') replacing (X, Y, σ), and output the corresponding items in X, Y as the intersection.

    (b) **[Hashing to bins]** The parties perform Figure 6.2 with parameters h, m, B, and randomly sampled hash functions $H_1, ..., H_h : \{0,1\}^{\sigma - \log_2 m} \to \{0,1\}^{\log_2 m}$ as input. Alice performs Step 1 of Figure 6.2 with set X to obtain $\mathfrak{B}_\text{X}$, and Bob performs Step 2 with Y to obtain $\mathfrak{B}_\text{Y}$.

2. **[Choose FHE parameters]** The parties agree on parameters (n, q, t) for an IND-CPA secure FHE scheme with circuit privacy. They choose t to be large enough so that $\log_2 t > \sigma - \log_2 m + \lceil \log_2 h \rceil + 1$.

3. **[Choose circuit depth parameters]** The parties agree on the windowing parameter $\ell \in [1, \log_2 B]$ and partitioning parameter $\alpha \in [1, B]$ as to minimize the overall cost.

4. **[Pre-process X]**

    (a) **[Partitioning]** Alice partitions its table $\mathfrak{B}_\text{Y}$ vertically (i.e. by columns) into α subtables $\mathfrak{B}_{\text{Y},1}, \mathfrak{B}_{\text{Y},2}, \ldots, \mathfrak{B}_{\text{Y},\alpha}$, each having $B' := B/\alpha$ columns.

    (b) **[Computing coefficients]** For each row v of each subtable, Alice replaces the row v with coefficients of the polynomial $\prod_s (x - v_s)$, i.e. it replaces v by $\text{Sym}(v) = ((-1)^j \sum_{S \subset [B'], |S| = j} \prod_{s \in S} v_s)_{0 \leq j \leq B'}$.

    (c) **[Batching]** For each subtable obtained from the previous step, Alice interprets each of its column as a vector of length m with elements in $\mathbb{Z}_t$. Then Alice batches each vector into m/n plaintext polynomials. As a result, the r-th subtable is transformed into $m/n \cdot B'$ polynomials $S_{i,j}^{(r)}$, $1 \leq i \leq m/n$, $0 \leq j \leq B'$.

5. **[Encrypt Y]**

    (a) **[Batching]** Bob interprets $\mathfrak{B}_\text{Y}$ as a vector of length m with elements in $\mathbb{Z}_t$. It batches this vector into m/n plaintext polynomials $\overline{Y}_1, ..., \overline{Y}_{m/n}$.

    (b) **[Windowing]** For each batched plaintext polynomial $\overline{Y}$ computed during Step 5a, Bob computes the component-wise $i \cdot 2^j$-th powers $\overline{Y}^{i \cdot 2^j}$, for $1 \leq i \leq 2^\ell - 1$ and $0 \leq j \leq \lfloor \log_2(B')/\ell \rfloor$.

    (c) **[Encrypt]** Bob uses FHE.Encrypt to encrypt each such power, obtaining m/n collections of ciphertexts $\{c_{i,j}\}$. Bob sends these ciphertexts to Alice.

6. **[Intersect]**

    (a) **[Homomorphically compute encryptions of all powers]** For each collection of ciphertexts $\{c_{i,j}\}$, Alice homomorphically computes a vector $\mathbf{c} = (c_0, \ldots, c_{B'})$, such that c_k is a homomorphic ciphertext encrypting $\overline{Y}^k$. In the end, Alice obtains m/n vectors $\mathbf{c}_1, \ldots, \mathbf{c}_{m/n}$.

    (b) **[Homomorphically evaluate the dot product]** Alice homomorphically evaluates

    $$\mathbf{r}_{i,r} = \sum_{j=0}^{B'} \mathbf{c}_i[B' - j] \cdot S_{i,j}^{(r)}\,, \quad \text{for all } 1 \leq i \leq m/n, \text{ and } 1 \leq r \leq \alpha,$$

    optionally performs modulus switching on the ciphertexts $\mathbf{r}_{i,r}$ to reduce their sizes, and sends them back to Bob.

7. **[Decrypt and get result]** For each $1 \leq r \leq \alpha$, Bob decrypts all ciphertexts it receives and concatenates the resulting m/n vectors into one vector $\mathfrak{R}_r$ of length m. Finally, Bob outputs

$$Y \cap X = \bigcup_{1 \leq r \leq \alpha} \{y \in Y : \mathfrak{R}_r[\mathsf{Loc}(y)] = 0\}\,.$$

Figure 6.4: Full FHE based PSI protocol.

Malicious behavior

When considering malicious behavior our protocol faces several challenges. Most notable is Alice's ability to compute an arbitrary function on Bob's homomorphically encrypted dataset. While Alice can not learn additional information directly from the ciphertexts, it is able to maliciously influence the correctness of the output, e.g. force the intersection/output to be Bob's full set, or more generally $f(X) \subseteq X$. Efficiently preventing such behavior by Alice appears to be extremely challenging.

For the case of a malicious receiver we need only to consider potential leakage which Bob can induce (sender has no output). First, Bob may provide a set of size greater than N_{X} due to its ability to fill vacant slots in the cuckoo hash table. Additionally, the argument that function privacy can easily be achieved through noise flooding no longer holds due to Bob being possibly providing ciphertexts with more noise than expected. As such, the noise level of the response ciphertexts may depend on Alice's set, and thereby leak additional information. However, in general we believe that this protocol provides reasonable protection against a malicious receiver for most practical applications. We leave a more formal analysis of the malicious setting and potential countermeasures to future work.

When receiver holds the larger set

So far we have made the assumption that Bob's set size is much smaller than Alice's set size. Here we remark that our protocol can be slightly modified to handle the opposite case, where Bob holds the larger set. The idea is that the two parties can perform our protocol with their roles switched until the last step. At this point, Bob (who has now been playing Alice's role) holds an encryption of a vector v. It samples a random plaintext vector r, and sends back to Alice an encryption of $v+r$. Alice decrypts this value, and sends back the plaintext vector $v + r$ to Bob, who can compute the final result v. This protocol is still secure in the semi-honest setting, and the communication remains linear in the smaller set and logarithmic in the larger set.

Name	n	q	t	DBC	κ
SEAL16384-1	16384	$2^{226} - 2^{26} + 1$	8519681	76	$\gg 128$ bits
SEAL16384-2	16384	$2^{226} - 2^{26} + 1$	8519681	46	$\gg 128$ bits
SEAL16384-3	16384	$2^{189} - 2^{21} + 9 \cdot 2^{15} + 1$	8519681	48	$\gg 128$ bits
SEAL8192-1	8192	$2^{226} - 2^{26} + 1$	8519681	46	≈ 120 bits
SEAL8192-2	8192	$2^{189} - 2^{21} + 9 \cdot 2^{15} + 1$	8519681	48	> 128 bits

Table 6.2: Encryption parameter sets for SEAL v2.1. Security estimates are based on [APS15, Alb17].

6.5 Implementation and Performance

6.5.1 Performance Results

We implemented our PSI protocol described in Figure 6.4. For fully homomorphic encryption we used SEAL v2.1 [KL16], which implements the Fan-Vercauteren scheme [FV12] in C++. The parameters for SEAL that we used are given in Table 6.2, along with their computational security levels κ, estimated based on the best currently known attacks [APS15, Alb17]. The column labeled “DBC” refers to the `decomposition_bit_count` parameter in SEAL. We note that these parameters are highly optimized for the particular computations that we perform.

We give detailed computational performance results for our protocol in Table 6.3 for both single and multi-threaded execution with 4, 16, and 64 threads. As Bob's computation is typically relatively small compared to Alice's, we restrict to single-threaded execution on Bob's side. Still, it is worth pointing out that also Bob's computation would benefit hugely from multi-threading, when available. Communication costs for our experiments are given in Table 6.4. We chose a statistical security level $\lambda = 40$, and a string length $\sigma = 32$ bits.

The benchmark machine has two 18-core Intel Xeon CPU E5-2699 v3 @ 2.3GHz and 256GB of RAM. We perform all tests using this single machine, and simulate network latency and bandwidth using the Linux `tc` command. Specifically, we consider a LAN setting, where the two parties are connected via local host with 10Gbps throughput, and a 0.2ms round-trip time (RTT). We also consider three WAN settings with 100Mbps, 10Mbps, and 1Mbps bandwidth, each with an 80ms RTT. All times are reported as the average of 10 trials.

Parameters			**Optim.**		**Running time (seconds)**									
N_{X}	N_{Y}	FHE parameters	α	ℓ	Sender pre-processing				Sender online				Receiver	
					$T = 1$	4	16	64	1	4	16	64	Enc.	Dec.
2^{24}	11041	SEAL16384-2	256	1	72.2	18.0	6.2	3.0	42.2	14.4	7.1	5.6	0.3	10.3
			128	2	70.9	19.1	6.3	3.1	38.9	15.6	9.8	9.1	0.5	5.1
		SEAL16384-1	64	3	76.8	20.6	6.7	3.3	41.1	21.6	16.2	16.9	0.9	2.6
	5535	SEAL8192-1	256	1	64.1	17.9	5.5	2.7	36.0	11.8	6.3	5.5	0.2	4.9
			128	2	71.2	18.5	6.3	2.9	36.1	14.2	9.6	9.2	0.3	2.4
			64	3	80.4	21.5	6.7	3.2	41.9	21.5	17.7	17.7	0.5	1.2
2^{20}	11041	SEAL16384-1	128	1	9.1	2.5	1.0	0.5	8.0	2.6	1.2	1.1	0.2	5.1
			64	2	6.9	2.0	0.8	0.4	5.2	1.8	1.1	1.0	0.3	2.7
			32	3	6.4	1.7	0.9	0.6	4.5	2.1	1.3	1.5	0.7	1.3
	5535	SEAL8192-2	128	1	5.1	1.4	0.6	0.4	4.2	1.5	0.8	0.7	0.1	1.9
			64	2	4.4	1.2	0.6	0.3	3.4	1.7	0.7	1.0	0.2	1.0
			32	3	4.3	1.2	0.5	—	3.6	1.4	1.5	—	0.3	0.5
2^{16}	11041	SEAL16384-3	16	1	1.2	0.3	0.2	—	1.3	0.6	0.6	—	0.2	0.5
			8	2	1.0	0.3	0.2	—	1.5	1.2	1.3	—	0.3	0.3
			4	3	0.9	0.3	—	—	1.9	1.7	—	—	0.5	0.1
	5535	SEAL8192-2	32	1	0.9	0.3	0.2	—	0.9	0.4	0.3	—	0.1	0.5
			16	2	0.7	0.2	0.1	—	0.7	0.3	0.3	—	0.1	0.2
			8	3	0.6	0.2	—	—	0.7	0.5	—	—	0.3	0.1

Table 6.3: Running time in seconds for our protocol with $T \in \{1, 4, 16, 64\}$ threads; $\lambda = 40$, $\sigma = 32$, $h = 3$. Since we implemented multi-threading by dividing the α partitions evenly between threads, having $T > \alpha$ offers no performance benefit. These cases are denoted by "—" in the table.

Pre-processing

The "Sender pre-processing" column in Table 6.3 measures the computational cost for Alice to prepare its coefficients of the polynomial $r \prod_{x \in X}(y - x)$, as mentioned in Section 6.3.3. More precisely, Alice's pre-processing work includes hashing and batching of its data, computing the coefficients in the right-hand side of (6.3), and sampling the random vectors. We also have Alice perform number theoretic transforms (NTT) to its plaintext polynomials to facilitate the underlying homomorphic multiplications in the second step described in Section 6.3.3.

We remark that our pre-processing can be done entirely offline without involving Bob. Specifically, given an upper bound on Bob's set size, Alice can locally choose parameters and perform the pre-processing. Upon learning Bob's actual set size, the parameters selected by Alice are communicated to Bob. We note that in order to achieve simulation-based security, the selected hash functions can only be used once. As such, each instance of the protocol must have an independent pre-processing phase, and in the event that a single pre-processing phase is used between several instances, an adversary with control of a party's set could force a hashing failure to occur. However, if such adversaries are not considered, then the pre-processing phase can be

Parameters			Optim.		Comm. size (MB)		Comm. time (seconds)			
N_X	N_Y	FHE parameters	α	ℓ	R → S	S → R	10 Gbps	100 Mbps	10 Mbps	1 Mbps
2^{24}	11041	SEAL16384-2	256	1	3.6	33.8	0.0	4.0	30.2	300.4
			128	2	6.3	16.9	0.0	2.4	19.0	186.7
		SEAL16384-1	64	3	12.7	8.4	0.0	2.2	17.4	169.4
	5535	SEAL8192-1	256	1	3.2	16.9	0.0	2.0	16.3	161.5
			128	2	4.1	8.4	0.0	1.3	10.3	101.0
			64	3	6.8	4.2	0.0	1.1	9.1	88.1
2^{20}	11041	SEAL16384-1	128	1	1.8	16.9	0.0	1.8	15.3	149.9
			64	2	3.6	8.4	0.0	1.3	9.9	98.0
			32	3	7.2	4.2	0.0	1.2	9.4	92.6
	5535	SEAL8192-2	128	1	1.1	8.4	0.0	1.0	7.8	77.0
			64	2	1.9	4.2	0.0	0.6	5.1	49.3
			32	3	3.4	2.2	0.0	0.6	4.7	45.0
2^{16}	11041	SEAL16384-3	16	1	2.3	2.1	0.0	0.5	3.6	35.5
			8	2	3.0	1.1	0.0	0.4	3.4	33.0
			4	3	6.0	0.5	0.0	0.6	5.4	52.9
	5535	SEAL8192-2	32	1	0.8	2.1	0.0	0.3	2.4	22.9
			16	2	1.5	1.1	0.0	0.3	2.2	20.7
			8	3	3.0	0.5	0.0	0.4	3.0	28.6

Table 6.4: Communication cost in MB for our protocol; $\lambda = 40$, $\sigma = 32$, $h = 3$. 10Gbps network assumes 0.2ms RTT, and the others use 80ms RTT. R → S and S → R denote the communications from receiver to sender, and from sender to receiver.

reused, resulting in significantly better performance.

PSI with longer items

When implementing our PSI protocol, we restrict the item length to be 32 bits. The reason is, although we can accommodate arbitrary size items in principle as described in Section 6.3.2, doing so naively with our protocol would require the encryption parameters to be substantially increased, which has a large negative impact on performance. We leave the task of making our protocol efficient for arbitrary size items to future work.

6.5.2 Comparison to Pinkas *et al.* [PSZ18]

Our primary point of comparison is the Pinkas *et al.* PSI protocol [PSZ18], in which the authors consider both the case of symmetric set sizes, and the setting where Bob's set is significantly smaller than Alice's. While our protocol can easily handle symmetric set sizes, our main advantage over [PSZ18] is in the asymmetric setting, which we now focus on. To make comparing the two protocols easier, we ran them on the same machine, and summarized the

Parameters		Protocol	Comm.	Total time (seconds)							
N_X	N_Y		Size (MB)	10 Gbps		100 Mbps		10 Mbps		1 Mbps	
				$T = 1$	4	1	4	1	4	1	4
2^{24}	11041	Us	23.2, †21.1	115.4	40.3	117.8	42.7	134.4	59.3	†290.8	†215.1
		[PSZ18]	480.9	40.5	23.3	88.0	66.4	449.5	427.5	4084.8	4067.2
		[KKRT16]	975.0	70.8	—	188.7	—	1269.1	—	12156.7	—
	5535	Us	20.1, †12.5, ‡11.0	105.2	34.8	107.2	36.7	†120.3	†45.8	†211.1	‡132.7
		[PSZ18]	480.4	40.1	23.1	87.9	65.5	449.2	427.3	4080.6	4064.3
		[KKRT16]	962.1	70.4	—	188.3	—	1263.5	—	12153.2	—
2^{20}	11041	Us	11.5	12.8	5.7	14.0	6.9	22.2	15.1	105.4	98.3
		[PSZ18]	30.9	3.3	2.1	7.0	5.6	29.8	28.3	263.7	262.1
		[KKRT16]	58.5	4.5	—	11.6	—	79.4	—	688.1	—
	5535	Us	5.6	8.6	3.3	9.2	3.9	13.3	8.0	53.6	48.3
		[PSZ18]	30.4	3.1	2.0	6.8	5.0	29.0	27.9	260.0	259.6
		[KKRT16]	57.3	4.4	—	11.5	—	79.3	—	686.0	—
2^{16}	11041	Us	4.1, †4.4	3.0	†1.7	3.4	†2.1	6.4	†5.3	36.0	35.0
		[PSZ18]	2.6	0.7	0.6	1.5	1.4	3.3	3.1	21.6	22.1
		[KKRT16]	4.5	0.4	—	1.4	—	5.6	—	48.2	—
	5535	Us	2.6	1.8	0.9	2.0	1.2	3.9	3.1	22.5	21.7
		[PSZ18]	2.1	0.7	0.6	1.4	1.3	2.9	2.8	19.8	21.3
		[KKRT16]	3.7	0.4	—	1.2	—	5.4	—	46.7	—

Table 6.5: Total communication cost in MB and running time in seconds comparing our protocol to [PSZ18] and to [KKRT16], with $T \in \{1, 4\}$ threads; $\lambda = 40$, $\sigma = 32$, $h = 3$. 10Gbps network assumes 0.2ms RTT, and others use 80ms RTT. Only single-threaded results are shown for [KKRT16] due to limitations of their implementation. The communication cost for [KKRT16] is based on the equation provided in their paper; empirical communication was observed to be ~ 1.5 times larger.

total running times side by side in Table 6.5. We chose to evaluate performance for the set sizes $N_Y \in \{5535, 11041\}$, $N_X \in \{2^{16}, 2^{20}, 2^{24}\}$ to maximize the utilization of ciphertext batching, described in Section 6.3.1. The sizes for N_Y were determined in Section 6.3.2 to be the largest that can guarantee a statistical security level of $\lambda \geq 40$. If a direct comparison to the running times reported in [PSZ18] is desired, the reader can feel free to round down our set sizes N_Y to match the sizes therein.

When comparing the two protocols, we find that our communication cost scales much better when Alice's set size is greater than 2^{16}. For instance, when considering strings of 32 bits, with $N_Y \leq 5535$ and $N_X = 2^{20}$, our protocol sends 5.6MB, while the same N_X, N_Y parameters applied to [PSZ18] result in 30.4MB of communication—a 5.4× improvement. Increasing N_X even further to $N_X = 2^{24}$, our protocol requires just 11.0MB of communication, whereas [PSZ18] requires over 480MB—a 43.7× improvement. Moreover, continuing to increase Alice's set size results in an even greater communication benefit.

When computing the intersection of sets of size $N_{\rm Y} \leq 5535$ and $N_{\rm X} = 2^{20}$ in a single-threaded LAN setting, our protocol requires 8.6 seconds. Evaluating the protocol of [PSZ18] using the same parameters results in an execution time of 3.1 seconds. While [PSZ18] is faster than our protocol in this particular setting, it also requires 5.4× more communication, and distributes the computational cost equally between the parties. That is, each party performs $O(N_{\rm X} + N_{\rm Y})$ operations. In contrast, our protocol places very few requirements on the computational power of Bob.

Since our protocol achieves a lower communication than [PSZ18] in the asymmetric set sizes setting, we obtain much better performance as we decrease the network bandwidth. To clearly demonstrate this, we consider several other network environments that model the WAN setting. In particular, we restrict the parties to a 100Mbps, 10Mbps, and 1Mbps networks with a 80ms round trip time. In these settings, our protocol outperforms [PSZ18] with few exceptions. Namely, the single-threaded 100Mbps setting, with $N_{\rm X} = 2^{24}, N_{\rm Y} \leq 5535$, our protocol requires 107.2 seconds, whereas [PSZ18] requires 87.9 seconds. However, our protocol receives a much greater speedup in the multi-threaded setting, reducing our running time to 36.7 seconds when the *sender* uses 4 threads. On the other hand, [PSZ18] requires 65.5 seconds for the same set sizes and with *both* parties using 4 threads—a nearly 1.8× slowdown compared to our protocol. As we further decrease the bandwidth, the difference becomes much more significant. In the 1Mbps single-threaded setting, with $N_{\rm X} = 2^{24}, N_{\rm Y} \leq 5535$, our protocol requires 211.1 seconds compared to [PSZ18] requiring 4080.6 seconds—a 19.3× improvement in running time. When utilizing 4 threads, our running time decreases to 132.7 seconds, while [PSZ18] requires 4064.3 seconds—a 30.6× improvement.

We also consider the running time of our protocol when more than 4 threads are used by Alice. When allowing 16 threads in the LAN setting, our running time decreases to 16.9 seconds for $N_{\rm X} = 2^{24}, N_{\rm Y} \leq 5535$. [PSZ18] on the other hand experiences less speedup over 4 threads, requiring just over 20 seconds for $N_{\rm X} = 2^{24}$ when performed with 16 threads. This demonstrates that our protocol can outperform [PSZ18] even in the LAN setting, when at least 16 threads are used by Alice.

An important property of our protocol is the relatively small amount of work required by Bob. In many applications the computations power of Bob is significantly less than Alice. This is most notable in the contact discovery application where Bob is likely a cellphone while Alice can be run at a large datacenter where computational power is inexpensive. For instance, Table 6.3 with parameters SEAL8192-1, $\alpha = 64, \ell = 3$ shows that for a intersection between 5535 and 2^{24} items, Bob need only perform 1.7 seconds of computation while the server with 16 threads required 18 seconds with a total of 11MB of communication, less than half the size of the average 2012 iOS application download size [Res12] and a tenth of the average 2015 daily US smartphone mobile data usage [Eri16]. In contrast, [PSZ18] requires 480MB of communication—a 44× increase–and the computational load of Bob is significantly higher requiring 50 million hash table queries and several thousand oblivious transfers.

6.5.3 Comparison to Kolesnikov *et al.* [KKRT16]

We also compare our protocol to that of Kolesnikov *et al.* [KKRT16], which optimizes the use of oblivious transfer. While their results do improve the running time for symmetric sets of large items, we found that when applied to our setting their improvement provides little benefit, and is outweighed by other optimizations employed by [PSZ18]. In particular, [PSZ18] considers a different oblivious transfer optimization which is more efficient on short strings, and also optimizes cuckoo hashing for the setting of asymmetric set sizes.

These design decisions result in [KKRT16] requiring 2× more communication than [PSZ18], and 87× more than our protocol, when intersecting 5535 and 2^{24} size sets with parameters SEAL8192-1, $\alpha = 64, \ell = 3$. When benchmarking [KKRT16], we found that the communication is actually $\sim$ 1.5 larger than their theoretical limit. The theoretical communication complexity of [KKRT16] is

$$N_{\text{X}} s v + k(1.2 N_{\text{Y}} + s)\,,$$

where $s = 6$ is the stash size in cuckoo hashing, $k \approx 444$ is the width of the pseudorandom code, $v = \lambda + \log_2(N_{\text{X}} N_{\text{Y}})$ is the size of the OPRF output, and 1.2 is related to cuckoo hashing utilization. The communication complexity of [PSZ18] also follows same equation, but with a smaller k due to more optimized oblivious transfer sub-protocol. Our protocol on the other hand requires

$$1.5 C \sigma N_{\text{Y}} \log_2 N_{\text{X}}$$

bits of communication, where C is a small constant for ciphertext expansion, $\sigma = 32$ is the string length, and 1.5 is related to the cuckoo hashing utilization with *no stash*. For example, when $N_{\text{X}} = 2^{24}$ and $N_{\text{Y}} = 5535$, our protocol requires only 12.5MB of communication, whereas the empirical communication of [KKRT16] in this setting is almost 115× larger.

This increase in communication translates into increased running times compared to [PSZ18] and our protocol in the WAN settings. For instance, when intersecting 5535 and 2^{24} items on a 10Mbps connection, our protocol is more than 57× faster, while [PSZ18] is only 3× faster. The total running times are summarized in Table 6.5 to make comparison to our protocol and to [PSZ18] easy. Since the implementation of [KKRT16] does not support multi-threading, we only present results for $T = 1$.

6.6 Conclusions

Although there has been huge progress in fully homomorphic encryption schemes since the groundbreaking work of Craig Gentry in 2009, it is still believed by many to be too expensive for practical use-cases. However, in this paper we have constructed a practical private set intersection protocol using the Fan-Vercauteren scheme, adopting and combining optimizations from both fully homomorphic encryption and cutting-edge work on PSI. We think our protocol is particularly interesting for the private contact discovery use-case, where it achieves a very low communication overhead: about 12MB to intersect a set of 5 thousand items with a set of 16 million items, which is significantly lower than in the previous state-of-the-art protocols. We regard our work as a first step to explore the possibilities of applying fully homomorphic encryption to private set intersection, and look forward to further discussions and optimizations.

Chapter 7

Improved PSI from Fully Homomorphic Encryption

Labeled PSI from Fully Homomorphic Encryption with Malicious Security by Hao Chen, Kim Laine & Peter Rindal, in CCS.

7.1 Introduction

Unbalanced PSI Most of the work on PSI has been designed for the *balanced* case, where the two sets are roughly of equal size, and the two parties have similar computation and storage capabilities. These protocols typically perform only marginally better when one of the sets is much smaller than the other. In particular, their communication cost scales at least linearly with the size of the larger set. In certain applications, however, Bob's set may be much smaller than Alice's. Bob might be a mobile device with limited battery, computing power, and storage, whereas Alice could be a high-end computing device. Moreover, the bandwidth between Bob and sender might be limited. This motivates the study of *unbalanced* PSI, where one set is much larger than the other. There have recently been several proposals optimizing for unbalanced PSI [CLR17, PSWW18, RA18]. Among these works [CLR17] achieves the smallest overall communication complexity, namely $O(|Y| \log |X|)$, where X denotes Alice's set and Y Bob's set, and $|X| \gg |Y|$. However, their results were limited to 32-bit items due to the significant performance overhead of extending to longer items. In this work we improve their protocol in terms of functionality, performance, and the security model.

Labeled PSI In certain applications, Alice holds a *label* ℓ_i for each item x_i in its set, and we wish to allow Bob to learn the labels corresponding to the items in the intersection. In other words, Bob should learn $\{(x_i, \ell_i) : x_i \in Y\}$ as a result of the protocol execution. When Bob's set Y consists of a single element, this is equivalent to the (single-server variant of) *Private Information Retrieval (PIR) by keywords* problem, considered first by [CGN97]. For ease of exposition, we will stick to the concept of Labeled PSI in the rest of the paper, noting that it is equivalent to a batched single-server symmetric PIR by keywords.

Labeled PSI has some immediate practical applications to private web service queries. For example, querying stock prices, location specific information, travel booking information, or web domain name information can reveal information to the service providers allowing them to conduct highly targeted price discrimination [Odl03], or to obtain sensitive personal or business information. Another example is a variant the private contact discovery problem [Mar14], where a user wishes to retrieve a public key for every person in her contact list who has registered to an instant messaging service for peer-to-peer communication. In this regard, Labeled PSI can provide the necessary functionality while guaranteeing query privacy.

7.1.1 Chapter Contributions

At a high level, our contributions can be summarized as follows:

- We use an OPRF preprocessing phase to achieve improved performance and security over [CLR17] by eliminating the need for an FHE scheme with circuit privacy.
- We build upon [CLR17] to support arbitrary length items by implementing a modified version of the generalized SIMD encoding algorithm of [HS14].
- We extend our protocol to achieve Labeled PSI with small communication complexity.

- We apply Labeled PSI to provide improved security against a malicious sender.

- We achieve full simulation-based security against a malicious receiver, improving upon [CLR17] which achieves security in the semi-honest model.

- We explain how to perform generic computation on the intersection and on the associated labels.

First, we improve upon the protocol of [CLR17] by leveraging a pre-processing phase, where the parties apply an Oblivious PRF (OPRF) to their input items. This change has two effects:

(i) Alice no longer needs to perform an expensive noise flooding operation on the result ciphertexts, as was necessary in [CLR17]. This allows the implementation to utilize more efficient FHE parameters, which further improves our performance and adds flexibility to the parametrization.

(ii) The pre-processing phase allow us to argue that our protocol is secure against a malicious receiver. In particular, we show that after the pre-processing phase the simulator can extract Bob's set and successfully simulate the rest of the protocol. We show that this pre-processing can either be performed using exponentiation [JL10, RA18] or oblivious transfer [OOS17, KKRT16]. Crucially, the former allows Alice to perform the pre-processing only once and reuse it over many protocol executions, which significantly improves performance in an amortized setting, e.g. in private contact discovery.

Second, all examples in [CLR17] were limited to performing comparisons of 32-bit items, whereas applications usually require longer items such as phone numbers, names, or public keys. The obvious extension of their protocol to fit larger items has suboptimal performance due to the significant performance overhead from using larger FHE parameters. We overcome this limitation by implementing a “lazy” version of the generalized SIMD encoding algorithm used in the HElib library [HS14], which allows encrypting fewer but longer items into a single ciphertext without requiring the FHE parameters to be increased.

Our third contribution is the design and implementation of a protocol for Labeled PSI. The challenge in this setting is how to allow Bob to learn a label ℓ_i for each item $x_i \in X \cap Y$, while still keeping the communication sublinear in $|X|$. Although many existing PSI protocols [RA18, PSWW18] can perform this functionality with a simple modification (encrypt the labels under their respective OPRF value), all of them fall short on the requirement that the communication be sublinear in the larger set. We propose a method which is similar to the original [CLR17] protocol, except that Alice evaluates an additional polynomial which interpolates the labels. When these labels are appropriately masked, Bob will be able to recover the label for exactly the items in the intersection.

Fourth, we discuss how a variant of Labeled PSI can be leveraged to achieve a reasonable notion of security against a malicious sender. For example, the protocol of [CLR17] suffers from an attack where Alice can force Bob to output its full set Y. In our protocol Bob will output $X \cap Y \cap \mathsf{leakage}(\widehat{Y})$, where $\mathsf{leakage}$ is specified by the malicious sender and is constrained in a reasonable way, and $\widehat{Y}$ denotes a hashing of the set Y (see Section 7.6.2). The high-level idea is to perform Labeled PSI where the label for an item z is $H(z)$ for some hash function H. We argue that the only efficient way for Alice to return the expected label for a receiver's item $y \in Y$ is to have the set X contain the item y. The exact assumption we make is that Alice can not homomorphically compute an encryption of $H(y)$ given an encryption of y and some pre-determined list of encryptions of powers of y, when H is a sufficiently complex hash function. The validity of this assumption depends on the difficulty of using leveled FHE to evaluate a circuit of depth higher than the pre-determined upper bound. Any efficient attack on this assumption would likely represent a very significant advancement in the state-of-the-art.

Fifth, we demonstrate how the output of the PSI computation can be secret-shared between the parties. This immediately allows for generic computation on the intersection using any general purpose MPC protocol. The idea behind this extension is for Alice to add an additional random value to all of the returned ciphertexts. When Bob decrypts them the two parties will hold an additive sharing of the comparison results. From such a sharing additional computation can be performed on the intersection. For example, the cardinality of the intersection, or the sum of the labels can be computed.

7.1.2 Summary of Notations

- X is the sender's set; Y is the receiver's set. We assume $|X| \gg |Y|$.
- σ is the length of items in X and Y.

- ℓ is the length of labels in Labeled PSI.
- n is the ring dimension in our FHE scheme (a power of 2); q is the ciphertext modulus; t is the plaintext modulus [FV12, DGBL+15].
- d is the degree of the extension field in the SIMD encoding.
- m is the cuckoo hash table size.
- α is the number of partitions we use to split the sender's set X in the PSI protocol (following [CLR17]).
- $[i, j]$ denotes the set $\{i, i+1, ..., j\}$, and $[j]$ is shorthand for the case $i = 1$.

7.2 The CLR17 Protocol

We now review the protocol of [CLR17] in detail. Following the architecture of [PSSZ15], their protocol instructs Bob to construct a cuckoo hash table of its set Y. Specifically, Bob will use three hash functions h_1, h_2, h_3, and a vector $B_R[0], \ldots, B_R[m]$ of $O(|Y|)$ bins. For each $y \in Y$, Bob will place y in bin $B_R[h_i(y)]$ for some i such that all bins contain at most one item. Alice will perform a different hashing strategy. For all $x \in X$ and all $i \in \{1, 2, 3\}$, Alice places x in bin $B_S[h_i(x)]$. Note that each bin on Alice's side will contain $O(|X|/m)$ items. It then holds that the intersection of $X \cap Y$ is equal to the union of all bin-wise intersections. That is,

$$X \cap Y = \bigcup_j B_R[j] \cap B_S[j] = \bigcup_j \{y_j\} \cap B_S[j]$$

where y_j is the sole item in bin $B_R[j]$ (or a special sentinel value in the case that $B_R[j]$ is empty). The protocol then specifies a method for computing $\{y\} \cap B_S[j]$ using FHE. Bob first sends an encryption of y, denoted as $[\![y]\!]$, to Alice who locally computes

$$[\![z]\!] := r \prod_{x \in B_S[j]} ([\![y]\!] - x)$$

When $y \in B_S[j]$, observe that one of the terms in the product is zero and therefore $[\![z]\!]$ will be an encryption of zero. $[\![z]\!]$ is returned to Bob, who concludes that $y \in X$ if $z = 0$. In the case that $y \notin B_S[j]$, the product will be the product of differences. Alice randomizes this product using a uniformly sampled element $r \in \mathbb{F}^*$ for some finite base field $\mathbb{F}$ used to encode the items. As a result, $z = 0$ if and only if $y \in X$. Otherwise z is uniformly distributed and independent of the set X.

Building on this general protocol, [CLR17] proposed several optimizations which make computing this circuit computationally efficient. First, recall that Bob has a vector of $m = O(|Y|)$ bins, each containing (at most) a single element $y_1, ..., y_m$. Each y_j must be intersected with $B_S[j]$. FHE naturally supports a technique which allows encrypting vectors and performing Single Instruction Multiple Data (SIMD) operations on the encrypted vectors ([SV14]). In this way many of the items y_j can be encrypted into a single ciphertext and processed concurrently, which results in a significant performance improvement.

Despite this, computing $[\![z]\!] := r \prod_{x \in B_S[j]}([\![y]\!] - x)$ directly was observed to be inefficient due to the performance penalty of using FHE to evaluate large degree polynomials on encrypted values. The multiplicative depth of directly computing z is $O(\log B)$ for $B \approx |X|/m$, and [CLR17] reduced it to $O(\log \log B)$ using a windowing technique. Namely, observe that z can be viewed as a polynomial $P(y) = a_B x^B + ... + a_1 x + a_0$ where the a_i are determined by r and $B_S[j]$. Alice needs to compute encryptions of all the powers of y between 1 and B. Given an encryption of only y, this can be done in $O(\log B)$ depth using the square multiply algorithm. However, Bob can now assist in the computation by sending additional powers of y. For example, if Bob sends encryptions of $y^{2^0}, y^{2^1}, y^{2^2}, ..., y^{2^{\log B}}$, Alice can use these terms to compute all necessary powers of y in multiplicative depth $O(\log \log B)$. They also partitioned Alice's bins into α subsets. Alice can then process each of these subsets independently. This reduces multiplicative depth further to $O(\log \log \frac{B}{\alpha})$. The downside of this approach is that for each y several response ciphertexts $z_1, ..., z_\alpha$ must be sent back to Bob, increasing the reply size by a factor of α. Finally, the authors pointed out that modulus switching to a smaller modulus can significantly improve this back-communication as no further computations on the ciphertexts is necessary.

7.3 PSI with Long Items

The [CLR17] protocol achieves good performance for 32-bit items and scales well to very large sets on Alice's side. However, it scales less well for longer items for the following reasons. Suppose the effective item length is σ bits. Then they need to set the plaintext modulus in the FHE scheme to $t \approx 2^\sigma$. Now let L denote the depth of the homomorphic evaluation at Alice's side. In [CLR17] the depth L depends double-logarithmically on $|X|$. Hence for our purposes we assume L is a constant. Then, the BFV scheme requires $\log q \gtrsim L \log t$ for correctness, but using the complexity estimates in Section 2.1.4 we see that the communication cost of [CLR17] grows linearly with σ; on the other hand, the computational cost grows *quadratically* with σ, which is undesirable.

Another side-effect of large σ comes from the security requirement: it drives up the FHE parameters t and q, and in order to keep the security level on par, the parameter n needs to increase as well. Now two cases can arise: if $|Y|$ is large compared to n then—since increasing n will increase the number of slots in each ciphertext—we end up using fewer ciphertexts, and the performance overhead is small; on the other hand, if $|Y|$ is of similar size as the previous n value then—after switching to new value of n—many slots could remain unused, which means the communication cost went up for no benefit.

Regardless of the initial length of the items, it is customary to apply a hash function of output length $\lambda + \log |X| + \log |Y|$, and perform the intersection protocol on these short hashes. Here λ denotes the statistical security parameter. In practice, we set $\lambda = 40$, thus we require t to be roughly 80 bits. However, using such a large value for t has a huge impact on the performance of the [CLR17] protocol. In this work, we resolve this issue by using the general SIMD encoding method proposed in [SV14], which allows coefficient-wise operations on vectors of flexible length and width. More precisely, for a tunable parameter d, we can operate on vectors of length n/d, where each entry can take t^d different values (when $d = 1$, this is the SIMD method used in [CLR17]).

More precisely, the plaintext space of BFV scheme equals $R_t = \mathbb{Z}_t[x]/(x^n+1)$. Suppose t is a prime number, and $x^n + 1 \pmod t$ factors into a product of irreducible polynomials $\{F_j(x)\}$, each of degree d. Moreover, suppose d is the smallest positive integer such that $t^d \equiv 1 \pmod{2n}$. Then the SIMD encoding can be explained as the following two isomorphisms

$$R_t \xrightarrow{\phi} \prod_j \mathbb{Z}_t[x]/(F_j(x)) \xrightarrow{\psi} \prod_{i=1}^{n/d} \mathbb{F}_{t^d}$$

where $\mathbb{F}_{t^d}$ is a fixed finite field of t^d elements. Now SIMD encoding corresponds to $\phi^{-1} \circ \psi^{-1}$, and decoding is the isomorphism $\psi \circ \phi$.

7.3.1 Trade-Offs for SIMD with Wider Slots

Communication cost The concrete communication cost of our PSI protocol can be computed in the following way: the query consists of $m/k \cdot (\log(|X|/m))$ ciphertexts, where $k = n/d$ is the number of slots supported in a ciphertext, and $m = O(|Y|)$. Each ciphertext has size $2n \log q = 2nL \log t$ bits. Here $\log t \approx \sigma/d$, since every element in the finite field $\mathbb{F}_{t^d}$ can represent a $d \log t$-bit item. The reply from Alice consists of $m/k \cdot \alpha$ ciphertexts, each with size roughly $2n \log(tn)$ bits.[1]

We view $|Y|$ (and m), $|X|$, n, and L as constants. Then the query size is a constant multiple of $\log d + \log |X| - \log n$, so increasing d will increase the query size. However, in our setup we usually have $\log |X| \gg \log n$. Since $d < n$ always, the $\log |X|$ term dominates. The reply size is also slightly worse for larger d, but its effect is small.

Computational cost Alice's online computation consists of $\Theta(|X|/k)$ homomorphic multiplications, each taking $n \log n (\log q)^2$ bit operations. After hiding all the constants, we see that the computational cost is $\Theta(1/d)$, so increasing d has a direct positive effect on the online computing time.

Effect of choosing different n Using a larger value of d has another implicit benefit: it allows choosing a smaller t for the same item bit-length, hence one can choose a smaller q, which also opens the possibility of choosing a smaller n. We can use the above heuristics to analyze the effect of changing n.

The computational cost is $|X|/d \log n \cdot L^2\sigma^2$ bit operations, and communication is

$$m(2d \log n + \sigma)\alpha + m \log(\frac{|X|d}{n}) \cdot 2L\sigma \,,$$

[1] This is because we can perform the modulus switching trick to reduce the modulus to $q' \approx tn$.

so both the computation and communication depend only logarithmically on the value of n. Therefore, its effect on performance is marginal when the other parameters are held fixed. We conclude that under the same setup, using wider slots (i.e. a larger value of d) to encode items result in larger communication and smaller on-line computation. On the other hand, the effect of the ring dimension n on overall performance is small if other parameters are held fixed.

7.3.2 Lazy SIMD Encoding Algorithm

[SV14] suggested that an FFT algorithm can be utilized for fast SIMD encoding and decoding. The FFT algorithm is very efficient when d is small, but for larger values of d there exist more efficient algorithms. For example, the HElib library [HS14] performs the encoding in two steps. Let $\mathbb{F}_j = \mathbb{Z}_t[x]/(F_j)$. They first map $(\mathbb{F}_{t^d})^k$ to $\mathbb{F}_1 \times \cdots \times \mathbb{F}_k$ through k field isomorphisms. Then a tree-based CRT algorithm is used to invert the first map ϕ: given $f_j(x) \in \mathbb{Z}_t[x]/(F_j)$, return $f \in \mathbb{Z}_t[x]/(F)$, such that $f_j = f \mod F_j$.

We make the observation that the second map ψ can sometimes be omitted in the encoding. Indeed, it is necessary if one wishes to homomorphically permute items in the $\mathbb{F}_{t^d}$ slots. Since our current application does not require such permutations, we could skip this step and solely use ψ and its inverse for decoding and encoding. This saves computation time as well as storage, since evaluating ψ requires the information of the isomorphisms $\mathbb{F}_j \to \mathbb{F}_{t^d}$.

On the other hand, the FFT algorithm can perform $\psi \circ \phi$ in one step, and there are fast algorithms that work with the $x^n + 1$ modulus. However, a natural way to utilize FFT in this scenario seems to require working with data of length n, using the $2n$-th roots of unity in $\mathbb{F}_{t^d}$. The complexity of such an algorithm is $O(n \log n)$ operations in $\mathbb{F}_{t^d}$.

To determine the optimal strategy, we performed a comparison by implementing both the lazy encoding algorithm and the FFT algorithm using FLINT [HJP13] and present the results in Figure 7.1. From the results we can see that the lazy encoding algorithm has a speed advantage which grows with the extension degree d.

$\log n$	t	d	FFT time (ms)	Lazy SIMD time (ms)
11	0x2E01	8	3.3	1.5
12	0x13FF	8	12	4
13	0x3401	16	32.6	9.2
14	0x2A01	64	512	23.5

Table 7.1: Comparison of SIMD encoding algorithms.

7.4 OPRF Pre-processing

We now demonstrate how a pre-processing phase can be performed to facilitate a more efficient online phase. The core idea is to first update the values being intersected using an oblivious PRF, where only Alice knows the key. This has the effect that several costly countermeasures employed by [CLR17] to protect Alice's set can be eliminated.

7.4.1 The CLR17 Approach

The [CLR17] protocol performs noise flooding to prove the security for Alice. The need for this stems from the fact that noise growth in homomorphic operations depends not only on the ciphertexts being operated on, but also on the underlying plaintexts. Thus, their security proof cannot work if the result ciphertexts are not re-randomized at the end, and if the underlying noise distribution is not hidden by flooding the noise by an appropriate number of bits.

There are at least two different problems with this approach. First, it requires Alice to estimate a heuristic upper bound on the size of noise, and ensure that there is enough noise room left to perform an appropriate amount of noise flooding. This makes it impossible to run their protocol with small FHE parameters, even for very small sets. Also, their protocol is fragile against malicious attacks. For example, Bob can insert more noise into its ciphertexts, causing Alice to noise-flood by fewer bits than it thinks. Now, by examining the noise distribution after the PSI computation, Bob can potentially obtain extra information about Alice's set.

7.4.2 Our Solution

We take a different approach to solving this problem, allowing us to get rid of noise flooding altogether. Namely, we use an OPRF to hash the items on both sides before engaging in the PSI protocol. This ensures that Alice's items $X \setminus Y$ are pseudo-random in Bob's view, preventing Bob from learning anything about the original items, even if it learns the hashed values in full.

Abstractly, we have Alice sample a key k and instruct it to locally compute $X' = \{F_k(x) \mid x \in X\}$. Bob then interactively applies the OPRF to its set, obtaining $Y' = \{F_k(y) \mid y \in Y\}$. From a security perspective it is now safe to send X' to Bob, who can infer the intersection from $X' \cap Y'$. However, this approach incurs a very high communication overhead, since X' can easily be over a gigabyte. Several recent works [PSWW18, RA18] try to tackle this problem by encoding X' in a compressed format, e.g. a Bloom filter or a cuckoo filter. However, the communication remains linear in the set $|X|$, and the compression can introduce false positives such as in the case of [RA18].

Our approach sidesteps this issue by applying an FHE-based PSI protocol to the sets X' and Y'. Overall, the communication complexity of our protocol is $O(|Y| \log |X|)$, as opposed to $O(|Y| + |X|)$ in the case of e.g. [PSWW18, RA18]. As previously described, we do not need to worry about noise flooding unlike [CLR17], since the OPRF already provides sufficient protection. This allows our protocol to use FHE parameters that are highly optimized, improving our performance and communication overhead.

More broadly, applying the OPRF to the sets also eliminates the need to perform two other procedures which protect Alice's set. First, recall that Alice performs simple hashing where its $|X|$ items are mapped to $O(|Y|)$ bins using three hash functions. In the original [CLR17] protocol all of these bins must then be padded with dummy items to an upper bound. This prevents some partial information from being leaked to Bob, e.g. m items hash to bin i, which implies that $|\{x \in X \mid h(x) = i\}| = m$. However, in the case that the OPRF is applied, the number of items in any given bin is a function of X', and therefore can be made public.

Secondly, the polynomials that Alice evaluates using Bob's set need not be randomized. Recall that in [CLR17] Alice evaluates homomorphically a polynomial of the form $F(y) = r \prod_{x \in X}(y - x)$, where $r \leftarrow \mathbb{F}^*$ is sampled uniformly at random each time the protocol is executed. This additional randomization was required to ensure that Bob does not learn the product of differences between y and X. It also has a significant impact on performance, as it increases the multiplicative depth by one. However, after the OPRF is applied, this polynomial is formed with respect to X'—not X—and therefore revealing the product of differences is no longer a security risk, since X' can securely be made public.

We consider two types of OPRFs which have different trade-offs. The first is a Diffie-Hellman based OPRF described by [JL10, RA18], which allows Alice to reuse OPRF values with many independent receivers, allowing for the cost of applying the OPRF to Alice's set to be amortized. Alternatively, an oblivious transfer based OPRF [OOS17, KKRT16] case be used, which is computationally more efficient, but can not be reused across several receivers.

7.4.3 DH-OPRF Pre-Processing

The Diffie-Hellman based OPRF protocol of [JL10] computes the function $F_\alpha(x) = H'(H(x)^\alpha)$, where H is a hash function modeled as a random oracle. This style of OPRF has been used several times in the context of PSI, e.g. in [Mea86, FIPR05, JL10, RA18]. In more detail, let G be a cyclic group with order q, where the One-More-Gap-Diffie-Hellman (OMGDH) problem is hard. H is a random oracle hash function with range $\mathbb{Z}_q^*$. Alice has a key $\alpha \in \mathbb{Z}_q^*$ and Bob has an input $x \in \{0,1\}^*$. Bob first samples $\beta \leftarrow \mathbb{Z}_q^*$ and sends $H(x)^\beta$ to Alice, who responds with $(H(x)^\beta)^\alpha$. Bob can then output $H'(H(x)^\alpha) = H'(((H(x)^\beta)^\alpha)^{1/\beta})$. The outer hash function H' is used to map the group element to a sufficiently long bit string, and helps facilitate extraction in the malicious setting.

In particular, by observing the queries made to $H(x_i)$, the simulator can collect a list of pairs $\{(x_i, H(x_i))\}$ which are known to Bob. From this set the simulator can compute the set $A = \{(x_i, H(x_i)^\alpha)\}$. For some subset of the $H(x_i)$, Bob sends $\{H(x_i)^{\beta_i}\}$ to the simulator, who sends back $\{H(x_i)^{\beta_i \alpha}\}$. For Bob to learn the OPRF value for x_i, it must send $H(x_i)^\alpha$ to the random oracle H'. At this time the simulator extracts x_i if $(x_i, H(x_i)^\alpha) \in A$. Although this OPRF does not facilitate extracting all x_i at the time the first message is sent, extraction is performed before Bob learns the OPRF value, which will be sufficient for our purposes.

In the context of our PSI protocol, this OPRF has the property that Alice can use the same key with multiple receivers. This allows Alice, who has a large and often relatively static set, to pre-process its set only once. This is particularly valuable since our protocol also allows for efficient insertions and deletions of data from the pre-processed set.

7.4.4 OT-OPRF Pre-Processing

An alternative approach is to use recent advances in Oblivious Transfer (OT) extension protocols [KKRT16, OOS17], which enable a functionality very similar to a traditional OPRF. The relevant difference is that Bob can only learn one OPRF output per key. This restriction mandates that the OPRF be used in a different way. In particular, we follow the PSZ paradigm [PSZ14, PSSZ15, PSZ18, KKRT16], where the OPRF is applied to the items after cuckoo hashing. First the parties perform cuckoo hashing, which ensures that Bob has at most one item per hash table bin. The parties then run an OT-based OPRF protocol, where Alice assigns a unique key to each bin. Bob updates the values in the cuckoo table with the OPRF outputs, while Alice similarly updates the values in its simple hash table. Note that Alice can learn an arbitrary number of OPRF outputs per key, which allows it to update all the values in each bin.

Another restriction with this approach is that Alice must pad its simple hash table with dummy items to ensure that Bob does not infer any partial information. That is, since the OPRF is applied after hashing, the number of items in any given bin is a function of X instead of the hashed set X'. It is therefore critical that the bins be padded to their maximum possible size, as was done in [CLR17].

As with the Diffie-Hellman based OPRF, Bob's input can be extracted. The exact details how how these protocols extract is quite involved, and we defer to [OOS17] for a detailed explanation.

7.5 Labeled PSI

We present two related approaches for enabling Labeled PSI. The first is compatible with the [CLR17] protocol, while the latter is optimized to take advantage of the OPRF pre-processing phase. This section will be presented in terms of Bob with a singleton set $\{y\}$ and obtaining the label ℓ_i if and only if Alice has a pair (x_i, ℓ_i) in its set for which $y = x_i$. The approaches naturally extend to the general setting by using cuckoo hashing on Bob's side.

7.5.1 Compatible With CLR17

We employ an idea of interpolation polynomials, also used in [FIPR05], to build our labeled PSI protocol. Recall that in the [CLR17] protocol the server homomorphically evaluates the polynomial $F(y) = r^* \prod_{x \in X}(x - y)$, where r^* is a random nonzero element in some finite field $\mathbb{F}$, and the coefficients of the polynomial $F(y)$ are elementary symmetric polynomials in the $x \in X$. It has the property that if $y \in X$, then $F(y) = 0$; otherwise $F(y)$ is a random element in $\mathbb{F}^*$. In the Labeled PSI case, Alice's input is a list of pairs $\{(x_i, \ell_i)\}_{i=1}^D$, where for simplicity we assume x_i and ℓ_i are elements of $\mathbb{F}$. Our goal is to construct a polynomial G, such that for any $y \in \mathbb{F}$

$$G(y) = \begin{cases} \ell_i & \text{if } y = x_i; \\ \text{random element in } \mathbb{F} & \text{otherwise.} \end{cases}$$

Note that there exists a polynomial $H(x)$ of degree less than $D = |Y|$, such that $H(y_i) = \ell_i$ for all $1 \leq i \leq D$. Then, we select $r \in \mathbb{F}$ randomly, and let

$$G(y) = H(y) + rF(y).$$

It is easy to verify that G has the desired property: if $y = x_i$, then $G(y) = H(x_i) = \ell_i$; if $y \notin X$, then since $F(y) \neq 0$ and r is random, we know that $G(y)$ is a random element in $\mathbb{F}$. At a high level, our protocol has Bob encrypt and send each of its items y using the FHE scheme; Alice evaluates the polynomials G and F homomorphically, and sends the results back. Bob then decrypts and obtains $(F(y), G(y))$, which is either equal to $(0, p_i)$ if $y = x_i$, or uniformly random in $\mathbb{F}^* \times \mathbb{F}$.

Note that in the above discussion, we implicitly assumed that labels are of the same size as the items. In case the labels are longer than the items, we can break down each label in chunks, and have the server repeat its computation several times. Finally, Bob can decrypt the parts of the label and re-assemble. The security proof is not affected, because in the case of a non-match, all the decrypted parts will be random strings.

Communication complexity We utilize the optimizations in [CLR17], with the modification that Alice homomorphically evaluates several polynomials instead of one. Hence, the communication complexity for our Labeled PSI is equal to $O(|Y|\log|X|\sigma + |Y|\ell)$, and the online computation complexity is $O(|X|\ell)$, where σ and ℓ denote the lengths of items and labels, respectively. Note that by hashing the items beforehand, we can assume $\sigma = \lambda + \log|X| + \log|Y|$, where λ is the statistical security parameter.

Computational complexity This Labeled PSI protocol introduces two additional computational tasks on top of the PSI protocol of [CLR17]. In the offline phase, Alice needs to interpolate a polynomial of degree $B' \approx \frac{|X|}{m\alpha}$. The Newton interpolation algorithm has complexity $O(B'^2)$, and is fast for small values of B'. The algorithm needs to be executed $m\alpha$ times, so the total complexity is $O(\frac{|X|^2}{m\alpha}) \cdot \ell/\sigma$. In the online phase, Alice needs to evaluate the interpolated polynomials homomorphically, which has a cost of $O(\frac{|X||Y|}{m^2} \cdot \ell/\sigma)$ FHE operations. Compared to the complexity of the PSI protocol in [CLR17], Alice's computation of labeled PSI grows by a factor of ℓ/σ, i.e. by the ratio of the label length and the item length.

7.5.2 OPRF Optimized Labeled PSI

If the parties perform the OPRF pre-processing phase, this procedure can be significantly improved. The core idea is to first encrypt all of the labels using the associated OPRF values as the key. All of these encrypted labels can then be sent to Bob, who uses the OPRF values for the items in its set to decrypt the associated labels. We stress that this approach requires no security guarantees from the homomorphic encryption scheme to ensure that information is not leaked to Bob about labels for items not in the intersection.

To avoid linear communication when sending these encrypted labels, we have Alice evaluate a polynomial which interpolates the encrypted labels, effectively compressing the amount of data that needs to be communicated. In more detail, Alice first computes $(x_i', x_i'') = OPRF_k(x_i)$ for all x_i in its set X. Here, x_i' will be used as the OPRF value for computing the intersection as before, while the second part x_i'' will be used to one-time-pad[2] encrypt the label as $\ell_i' = \ell_i + x_i''$. Alice then computes a polynomial G with minimal degree such that $G(x_i') = \ell'$.

[2] Note that x_i'' can be extended to an arbitrary size ℓ_i using a PRG.

One of the main advantages of this approach is that the degree of the online computation is reduced due to G not requiring additional randomization. Recall from above that the result of evaluating the symmetric polynomial F has to be randomized by multiplying with a nonzero r. This increases the degree of the computation by one, which can require larger FHE parameters.

Looking forward, our approach for improved security against a malicious sender requires the use of Labeled PSI, but interestingly does not require the evaluated symmetric polynomial $F(y)$ to be sent back to Bob. As such, by not requiring F in the computation of the labels, we gain an addition performance improvement in the malicious setting by not computing or evaluating F.

7.5.3 Full Protocol

We present our full protocol for labeled PSI in Figure 7.2.

7.6 Malicious Security

We will now show that our protocol is secure against a malicious receiver, while providing privacy against a malicious sender. Moreover, we will characterize the set of attacks that Alice can perform.

7.6.1 Malicious Receiver

With the addition of the OPRF pre-processing phase, it is relatively straightforward to show that our protocol achieves full security in the presence of a malicious receiver. First, Bob performs $|Y|$ OPRF invocations and receives the corresponding values. From this, Bob homomorphically encrypts its queries, and sends it to Alice. At this point the simulator extracts Bob's input Y, and forwards it to the ideal functionality, which responds with $X^* = X \cap Y$. The simulator pads X^* with random values not in Y to the size of $|X|$, and then inserts X^* into the simple hash table as would be done for Y. Finally, the simulator completes the protocol. The correctness of the simulator directly follows the proof of [JL10]. In particular, all OPRF values apart from the $|Y|$ which were extracted are uniformly distributed in Bob's view. As such, padding X^* with uniformly distributed values induces

Input: Receiver inputs set $Y \subset \{0,1\}^*$ of size N_{Y}; sender inputs set $X \subset \{0,1\}^*$ of size N_{X}. $N_{\mathrm{X}}, N_{\mathrm{Y}}$ are public. κ and λ denote the computational and statistical security parameters, respectively.
Output: Bob outputs $Y \cap X$; Alice outputs $\perp$.

1. [**Sender's OPRF**] Alice samples a key k for the [JL10] OPRF $F : \{0,1\}^* \to \{0,1\}^\kappa$. Alice updates its set to be $X' = \{H(F_k(x)) : x \in X\}$. Here H is a random oracle hash function with a range of $\sigma = \log_2(N_{\mathrm{X}} N_{\mathrm{Y}}) + \lambda$ bits which is sampled using a coin flipping protocol.
2. [**Hashing**] The parameter m is agreed upon such that cuckoo hashing N_{Y} balls into m bins succeed with probability $\geq 1 - 2^{-\lambda}$. Three random hash function $h_1, h_2, h_3 : \{0,1\}^\sigma \to [m]$ are agreed upon using coin flipping. Alice inserts all $x \in X'$ into the sets $\mathcal{B}[h_1(x)], \mathcal{B}[h_2(x)], \mathcal{B}[h_3(x)]$.
3. [**Choose FHE parameters**] The parties agree on parameters (n, q, t, d) for an IND-CPA secure FHE scheme. They choose t, d to be large enough so that $d \log_2 t \geq \sigma$.
4. [**Choose circuit depth parameters**] The parties agree on the split parameter $B < O(|Y|/m)$ and windowing parameter $w \in \{2, \ldots, \log_2 B\}$ as to minimize the overall cost.
5. [**Pre-Process** X]
    - (a) [**Splitting**] For each set $\mathcal{B}[i]$, Alice splits it into α subsets of size at most B, denoted as $\mathcal{B}[i,1], \ldots, \mathcal{B}[i,\alpha]$.
    - (b) [**Computing Coefficients**]
        - i. [**Symmetric Polynomial**] For each For each set $\mathcal{B}[i,j]$, Alice computes the symmetric polynomial $S_{i,j}$ over $\mathbb{F}_{t^d}$ such that $S_{i,j}(x) = 0$ for $x \in \mathcal{B}[i,j]$.
        - ii. [**Label Polynomial**] If Alice has labels associated with its set, then for each set $\mathcal{B}[i,j]$, Alice interpolates the polynomial $P_{i,j}$ over $\mathbb{F}_{t^d}$ such that $P_{i,j}(x) = \ell$ for $x \in \mathcal{B}[i,j]$ where ℓ is the label associated with x.
    - (c) [**Batching**]

      View the polynomials $S_{i,j}$ as a matrix where i indexes the row. For each set of n/d rows (non-overlapping and contiguous), consider them as belonging to a single *batch*. For b-th batch and each j, take the k-th coefficient of the n/d polynomials, and batch them into one FHE plaintext polynomial $\overline{S}_{b,j}$.

      For Labeled PSI, perform the same batching on the label polynomials $P_{i,j}$ to form batched FHE plaintext polynomials $\overline{P}_{b,j}$.

Figure 7.1: Full Labeled PSI protocol (sender's offline pre-processing).

the same distribution. Compared to [JL10], the primary difference in our protocol is the use of FHE to compress the amount of communication.

7. [**Encrypt** Y]

    (a) [**Receiver's OPRF**] Bob performs the interactive OPRF protocol of [JL10] using its set Y as private input. Alice uses the key k as its private input. Bob learns $F_k(y)$ for $y \in Y$ and set $Y' = \{H(F_k(y)) : y \in Y\}$.

    (b) [**Cuckoo Hashing**] Bob performs cuckoo hashing on the set Y' into a table $\mathcal{C}$ with m bins using h_1, h_2, h_3 has the hash functions.

    (c) [**Batching**] Bob interprets $\mathcal{C}$ as a vector of length m with elements in $\mathbb{F}_{t^d}$. For the bth set of n/d (non-overlapping and contiguous) in $\mathcal{C}$, Bob batches them into a FHE plaintext polynomial $\overline{Y}_b$.

    (d) [**Windowing**] For each $\overline{Y}_b$, Bob computes the component-wise $i \cdot w^j$-th powers $\overline{Y}_b^{i \cdot 2^j}$, for $1 \leq i \leq w-1$ and $0 \leq j \leq \lfloor \log_w(B) \rfloor$.

    (e) [**Encrypt**] Bob uses $\mathsf{FHE.Encrypt}$ to encrypt each power $\overline{Y}^{i \cdot 2^j}$ and forwards the ciphertexts $c_{i,j}$ to Alice.

8. [**Intersect**] For the bth batch,

    (a) [**Homomorphically compute encryptions of all powers**] Alice receives the collection of ciphertexts $\{c_{i,j}\}$ and homomorphically computes a vector $\mathbf{c} = (c_0, \ldots, c_\alpha)$, such that c_k is a homomorphic ciphertext encrypting $\overline{Y}_b^k$.

    (b) [**Homomorphically evaluate the dot product**] For each $\overline{S}_{b,1}, ..., \overline{S}_{b,\alpha}$, Alice homomorphically evaluates

    $$z_{b,j} = \sum c \cdot \overline{S}_{b,j}$$

    and optionally performs modulus switching on the ciphertexts $z_{b,j}$ to reduce their sizes. All $z_{b,j}$ are sent back to Bob. If Labeled PSI is desired, repeat the same operation for $\overline{P}$ and denote the returned ciphertexts $q_{b,j}$.

9. [**Decrypt and get result**] For the b-th batch, Bob decrypts the ciphertexts $z_{b,1}, ..., z_{b,\alpha}$ to obtain $r_{b,1}, ..., r_{b,\alpha}$, which are interpreted as vectors of n/d elements in $\mathbb{F}_{t^d}$.

    Let $r_1^*, ..., r_\alpha^*$ be vectors of m elements in $\mathbb{F}_{t^d}$ obtained by concatenating $r_j^* = r_{1,j}^* || ... || r_{md/n,j}^*$. For all $y' \in Y'$, output the corresponding $y \in Y$ if

    $$\exists j : r_j^*[i] = 0\,,$$

    where i is the index of the bin that y' occupies in $\mathcal{C}$.

    If Labeled PSI is desired, perform the same decryption and concatenation process on the $q_{b,j}$ ciphertexts to obtain the m element vectors $\ell_1^*, ..., \ell_\alpha^*$. For each $r_j^*[i] = 0$ above, output the label of the corresponding y to be $\ell_j^*[i]$.

Figure 7.2: Full Labeled PSI protocol continued (online phase).

7.6.2 Malicious Sender

Achieving full simulation based security against a malicious sender is extremely challenging in our setting. Arguably, the most significant barrier is that we require the communication complexity to be *sublinear* in the size of Alice's set. This makes traditional methods for extracting their set, e.g. ZK proofs [JL10], not viable due the associated communication overhead being linear.

The second biggest challenge is to enforce that Alice performed the correct computation. In our protocol Alice can deviate from the prescribed protocol in numerous way. For instance, it is well known [FIPR05, RR17a] that Alice can incorrectly perform simple hashing, which allows Bob's output distribution to depend on Bob's full set. Moreover, in our protocol the situation is even worse in that Alice obtains a homomorphically encrypted copy of Bob's set. Instead of using it to evaluate our PSI circuit, Alice has a large degree of freedom to compute a different circuit, which may arbitrarily depend on Bob's set.

For example, it is trivial for Alice to force Bob to output their full set. Recall that in [CLR17], and in our semi-honest protocol, when the returned ciphertext contains a zero, Bob interprets this to mean their corresponding item is in the intersection. A straightforward attack for Alice would then be to simply encrypt and return a vector of zeros if it holds a public key, or otherwise return an all zero ciphertext.[3] Bob would then conclude that its full set is the intersection.

Faced with these significant challenges, we choose to forgo simulation based security when in the presence of a malicious sender. Instead, we show that (1) our basic protocol achieves privacy against a malicious sender; (2) a simple modification of our protocol can significantly restrict the class of attacks that a malicious sender can perform.

Privacy

This notion of privacy against a malicious sender and full security against Bob has previously been considered in the context of PSI by Hazay and Lindell

[3] The homomorphic encryption library used may or may not reject such a ciphertext as invalid. If it does, Alice can still multiply any ciphertext obtained from Bob by a nonzero plaintext mask that sets all but one vector slot value to zero, leading Bob to interpret all but one item as matching.

[HL10]. Informally, privacy guarantees that Alice learns nothing about that receiver's input. This is a relaxation of simulation based security in that, for example, we do not achieve independence of inputs, nor guarantee output correctness. For a formal description we refer the reader to [HL10, Definition 2.2].

Showing that our protocol achieves privacy is relatively straightforward since Alice receives no output. Conceptually, we can divide our protocol into two phases: the OPRF pre-processing phase, and the main intersection phase. Privacy in the pre-processing phase naturally follows from the security of OPRF. That is, Alice can not learn any partial information about Bob's inputs. Similarly, all messages that are received by Alice in the main phase are homomorphically encrypted, and therefore guarantee that no information is revealed.

Restricting Attacks

First we consider the setting where Alice does not reuse the result of the pre-processing phase with multiple receivers. Here our technique for restricting the types of attacks Alice can perform is inspired by Labeled PSI combined with the fact that evaluating a high-depth circuit using leveled FHE which supports much smaller depth is hard—if not impossible. In particular, for Bob to conclude that an item y is in the intersection, we require that Alice return an encryption of the label $z = H(\mathrm{OPRF}_k(y))$, where H is a sufficiently complex hash function which we model as a random oracle, e.g. SHA256. Intuitively, Alice has two options for computing z: (1) it must know some $x = y$ and locally compute $z = H(\mathrm{OPRF}_k(y))$; (2) it must use Bob's encrypted set of $\{\mathrm{OPRF}_k(y)\}$to homomorphically evaluate a circuit that computes H using the given leveled FHE scheme under the fixed encryption parameters that the two parties have agreed to use.

We heuristically argue that evaluating such a circuit is extremely difficult—if not impossible—using leveled FHE, where the parameters are chosen to support a much smaller multiplicative depth. While there is no guarantee that a high degree polynomial can not be evaluated, we experimentally find that for our parameters the noise level always overflows when even a few extra multiplications are performed. In addition, hash functions such as SHA256 have very high depth, and seem extremely difficult to evaluate using arithmetic FHE schemes due to the switching between binary and modular arithmetic (in SHA256). Moreover, the depth of H can be increased arbitrarily by repeatedly applying the hash function, and the number of repetitions can be decided after the encryption parameters have been selected.

Under the assumption that evaluating H is infeasible under the given (leveled) FHE scheme, Alice must know an encryption of $\mathrm{OPRF}_k(y)$ to be able to compute an encryption of $H(\mathrm{OPRF}_k(y))$. As such, Alice is restricted to choosing a set X' of polynomial size, possible larger than $|X|$, and using this in the PSI protocol. We note that Alice is not committed to its set X, and can use an arbitrary subset of it in each protocol invocation. However, this does imply that for a receiver's set Y with high entropy, e.g. a set of public keys, Alice has no efficient way to influence the intersection to contain an item in $Y \setminus X$. Hence the intersection will be of size at most $X \cap Y$.

On the other hand, Alice is still able to make the intersection indirectly depend on the set $Y \setminus X$. In particular, Alice can choose a low-depth circuit $\mathsf{leakage}(\cdot)$ and is able to influence the intersection to be

$$X \cap Y \cap \mathsf{leakage}(\widehat{Y})\,,$$

where $\widehat{Y}$ is Bob's cuckoo hash table containing the OPRFvalues for its set Y, and additionally certain powers of these values if windowing is used. This leakage function models the fact that Alice can perform some malicious computation with Bob's encrypted hash table. This allows Alice to conditionally remove items from the intersection based on items which are in $Y \setminus X$. While such an attack can be serious in some settings, this leakage is significantly less than in [CLR17], where Alice can force the intersection to be Y.

Now we turn our attention to the setting where Alice reuses the pre-processing phase with multiple receivers. Here the hashing parameters have to be fixed and reused. In particular, the cuckoo hash function and the hash function H used to hash to σ bit strings are picked by Alice. This deviates from the specification in Figure 7.2 steps (1) and (2), where the parties jointly sample random hash functions. This has the implication that Alice can select hash functions which conditionally fail based on Bob's set Y. For example, if $y, y' \in Y$ both hash to cuckoo position i under all three hash function, cuckoo hashing will fail. This failure would be observable by Alice, and leak that Bob's set is a member of all the sets which fail to cuckoo hash under these parameters—a single bit of information. While this particular attack may happen with negligible probability, it is possible that a more sophisticated selective failure attack could succeed with noticeable probability.

While such attacks can be serious, we argue that many applications can tolerate leaking a single bit. One countermeasure which can be employed is to sample the hash functions from a public reference string. This could significantly restrict the effectiveness of such selective failure attacks, as the hash functions are then fixed.

7.7 Experiments

We implemented our protocols (unbalanced PSI for arbitrary length items, and Labeled PSI) and benchmarked against previous methods. For unbalanced PSI with both long and short items our points of comparison are [RA18, BBDC+11] and [CLR17] respectively. For labeled PSI, we compare with multi-PIR by keyword from the multi-query SealPIR solution of [ACLS17].

Our implementation is built from scratch on top of the homomorphic encryption library SEAL v2.3.0-4, which is based on the BFV scheme [FV12]. We give a detailed report of the end-to-end and online running times along with the communication overhead of our protocol in Figure 7.2, both in single and multi-threaded settings. We restrict Bob to at most 4 threads to model a low power device, while Alice utilizes up to 32 threads as denoted in the table. Figure 7.3 shows a comparison with the unbalanced PSI protocols of [RA18, BBDC+11, CLR17].

We benchmark the protocols on a 32-core Intel Xeon CPU with 256 GB of RAM. We note that this machine is similar to that utilized by [CLR17], and that the numbers reported for their protocol are obtained directly from their paper. All protocols are ran in the LAN setting with a 10 Gbps throughput, and sub-millisecond latency.

7.7.1 Unbalanced PSI

Figure 7.2 contains our main set of performance numbers and demonstrate a wide flexibility in set sizes. For Alice, we consider set sizes of 2^{20}, 2^{24} and 2^{28}, while Bob's set sizes range between 128 and 4096. We note that for each of Bob's set sizes, approximately 1.33 times more items could be added with no difference in performance due to extra space in the cuckoo table. However, to give a fair comparison with other protocols without parameter restrictions we round down to the nearest power of two.

We begin with our performance numbers for the smallest set sizes of $|X| = 2^{20}$ and $|Y| = 128$. For such a small intersection our protocol is extremely efficient, requiring an online time of less than a second on a single thread, and only 4.7 MB of communication. When Bob's set is increased to 512 items we observe only a minimal increase in running time and communication. Moving to our largest receiver set size of 4096, we observe roughly a 4× increase in online running time and communication. This sublinear linear growth in overhead with respect to Bob's set size is attributed to the ability to use more efficient FHE parameters.

With respect to Alice's offline running time, we observe that in almost all cases with $|X| = 2^{20}$ the running time is roughly 40 seconds on a single thread. The one exception is for $|Y| = 256$, which has double the running time. This is attributed to the FHE parameters used that allowed an efficient online time at the expense of increased computation during the offline phase.

Increasing Alice's set size to $|X| = 2^{24}$ we observe a similar trend, where a smallest set sizes for Bob all obtain the same performance. Indeed, for $|Y| = 128, 256, 512$ the same FHE parameters were utilized, which yield a single thread online running time of 9.1 seconds, and 11.2 MB of communication. The choice to use oversized parameters for the smaller set sizes stems from the highly complex interplay between the parameters that can be optimized, while also maintaining a computational security level of128 bits. We refer to Section 7.3.1 for a more detailed explanation.

For a receiver's set size of 4096 we observe an online running time of 22 seconds in the single thread setting, and 24.6 MB of communication. Alice's offline time required 807 seconds on a single thread, or 32 seconds on 32 threads. Interestingly, this offline running time is faster as Alice's set size increases. Among other things, this is the result of Alice's database having fewer items per bin as $|Y|$ increases, which in turn decreases the degree of the polynomials Alice needs to compute in pre-processing, resulting in improved performance.

In addition, we consider the case of $|X| = 2^{28}$ which, as far as we are aware of, is the largest PSI set size to have ever been considered in the two party setting, and is only surpassed in a weaker model where an untrusted third party assists in the computation [KMRS14]. For this case we consider a receiver's set size of 1024, and observe that the online phase can be performed

$\|X\|$	$\|Y\|$	Sender offline				Online				Comm.	
		T=1	4	8	32	T=1	4	8	32	$R \rightarrow S$	$R \leftarrow S$
2^{28}	2048	–	–	–	4,628	–	–	–	28.5	23.6	10.2
	1024	–	–	–	4,628	–	–	–	12.1	16.4	10.2
2^{24}	4096	806	206	111	32	22.0	5.9	3.5	2.2	17.4	7.2
	2048	747	483	58	18	12.6	3.4	2.0	1.4	16.4	8.1
	1024	1430	418	155	51	17.7	5.2	2.9	1.1	6.1	5.1
	512	1368	267	146	45	9.1	2.6	1.5	0.7	6.1	5.1
	256	1368	267	146	45	9.1	2.6	1.5	0.7	6.1	5.1
	128	1368	267	146	45	9.1	2.6	1.5	0.7	6.1	5.1
2^{20}	4096	43	6.1	3.3	1.2	4.2	1.3	1.3	1.2	8.4	7.2
	2048	39	4.9	2.7	1.1	2.1	0.7	0.6	0.6	5.7	3.6
	1024	40	5.1	2.9	1.1	2.0	0.7	0.5	0.5	4.9	2.5
	512	36	5.5	2.6	0.5	1.0	0.4	0.3	0.3	2.9	2.5
	256	87	18.4	9.5	3.3	2.4	0.8	0.6	0.3	4.2	1.0
	128	46	6.7	3.8	1.2	0.9	0.4	0.3	0.2	1.7	3.0

Table 7.2: Performance metrics of our protocol in the LAN setting for various set sizes. "Sender offline" reports the running time in seconds required to initialize Alice's set. This is non-interactive and can be reused with multiple receivers. "Online" reports the end-to-end running time in seconds required to perform the intersection, where the "Comm." columns report the directional communication requirements in MB. T is the number of threads used by Alice. Bob uses $\max\{T, 4\}$ threads.

in just 12 seconds when Alice and receiver respectively use 32 and 4 threads. At the same time, the online phase utilizes only 35 MB of communication. The primary impact of such a large set size is on Alice's offline running time. However, in cases where such a large set is used, it is highly likely that the set is held by a powerful server and is relatively static, in which case Alice can amortize the cost of the pre-processing across several protocol executions. We also note that our protocol allows fast additions and deletions from the pre-processed set by updating only small targeted locations when necessary. Moreover, the current implementation of the offline phase is far from optimal in that there exists more efficient algorithms for computing coefficients of Alice's polynomials, which is the primary bottleneck.

7.7.2 Comparison

We now move on to the comparison with the OPRF-based protocols of [RA18, BBDC+11] and the FHE-based protocol of [CLR17]. First we recall the protocol paradigm of [RA18, BBDC+11, JL10]. These protocol utilize the same Diffie-Hellman based OPRF as our protocol, where Alice holds the key k and applies the OPRF to its set X to obtain $X' = \{F_k(x) : x \in X\}$. Bob then interactively computes $Y' = \{F_k(y) : y \in Y\}$. Next Alice sends X' to Bob, who infers the intersection from $X' \cap Y'$ [JL10].

The main limitation of this approach is that Bob requires communication linear in the larger set X. The protocol above was first described in the malicious setting by [JL10], and later optimized for the semi-honest setting by [BBDC+11]. For modest size X this approach works reasonably well. For instance, with $|X| = 2^{20}$ and a receiver's set of 5535 items, the communication of X' is 10 MB. However, increasing $|X|$ further to 2^{24} or 2^{28} starts to become less practical with 160 MB and 2.6 GB of communication, respectively.

One of the most compelling properties of this paradigm is that after the communication of X', Bob can check the membership of a y in X using $O(\kappa)$ communication and computation—effectively a constant. Observing this, [RA18, KLS+17] both suggest that X' be sent during an offline or preprocessing phase, and then the online phase can have linear overhead in the smaller set, which provides extremely good performance. Moreover, many adaptive set intersections with X can be performed, where Bob is able to use a different set each time. The online communication and running time of this approach remains linear in Bob's set size.

On top of this general paradigm, [RA18, KLS+17] suggest that the encoded set X' can be compressed using techniques such as a cuckoo filter or a Bloom filter. [RA18] showed that for certain parameters a cuckoo filter can reduce the communication by roughly 3×, bringing the communication for $|X| = 2^{24}$ and 2^{28} down to 48 and 806 MB, respectively. This approach has the disadvantage that such a large amount of compression introduces a relatively high false positive rate of $2^{-13.4} \times |Y|$. That is, if Bob has 1024 items outside the intersection, the probability that Bob will output a wrong item is $2^{-3.2}$, which is extremely high for cryptographic protocols. In contrast, our protocol and [CLR17] both provide a false positive rate of at least 2^{-40}, and this rate is independent of Bob's set size.

In addition, the communication complexity of [RA18] remains linear in the larger set size. In the example of $|X| = 2^{28}$ the communication of 804 MB can be prohibitive for many applications. Moreover, for this online-offline technique to work, Bob must store the compressed set long-term, which can be prohibitive on mobile devices. In contrast to this paradigm, our FHE-based protocol achieves communication complexity which is sub-linear in the larger set size and requires no offline storage by Bob. When considering $|X| =$

2^{28} and $|Y| = 1024$, the total communication is 27 MB compared to 804 MB: a 30× improvement in communication and a 2^{26} times improvement in the false positive rate. To achieve an equivalent false positive rate, e.g. in the [BBDC+11] protocol, the communication increases to 2.6 GB—a difference of 100×.

Finally, there is a difference in the security achieved with respect to the information revealed when X is updated. In the case of [RA18, BBDC+11, KLS+17], Bob learns exactly how many items we inserted and removed from X. Even worse, the deletion of an item from X can not be enforced, since Bob can simply keep the corresponding OPRF value which they hold in X'. As such, deletions can not be enforced even in the semi-honest model. In contrast, our protocol can easily add and remove items with near perfect security, given that the returned polynomials are randomized as done in the [CLR17] protocol. In particular, the issue with deleting an item from X does not arise since only Alice holds X.

With respect to [CLR17], we compare quite favorably in many aspects. First recall that [CLR17] is practically restricted to 32-bit items, while we can support arbitrary length items. In particular, in [CLR17] the items being compared within FHE are actually only 22-bits, but are extended to 32 bits using a hashing technique known as phasing [PSSZ15]. Our protocol contrasts this by comparing 80-bit items within the FHE computation. When using the "hash to smaller domain" techniques of [PSZ14], this is sufficiently large to effectively support arbitrary length items, while achieving 40 bits of statistical security.

Given this significant improvement, our protocol still achieves a similar communication overhead, and often better online running times. For example, when comparing $|X| = 2^{24}$ and $|Y| = 5535$, our protocol requires 25 MB of communication and 22 seconds in the online phase on a single thread, while [CLR17] requires 20 MB and 40 seconds. This is almost a 2× improvement in running time and the ability to compare arbitrary length strings at the expense of a slight increase in communication. Moreover, when Bob has an even smaller set, our protocol is able to scale the FHE parameters even smaller, which allows less communication. For example, when Bob has 512 or 1024 items, our protocol requires 9.1 and 17.7 seconds, respectively, and just 11.2 MB of communication. On the other hand, due to the noise flooding performed by [CLR17], they are unable to take advantage of more efficient FHE parameters while also maintaining 128 bits of computational security. As such, our protocol can be 2 to 4 times faster than [CLR17], and send almost half the amount of data, while at the same time supporting arbitrary

$\lvert X\rvert$	$\lvert Y\rvert$	Protocol	Sender Offline	Offline Comm. & Receiver Storage	Online Time	Online Comm.
2^{28}	2048	Ours*	4,628	0	28.5	34.05
		[RA18]*	182	806	0.1	0.13
		[BBDC+11]*	182	2,684	0.1	0.13
	1024	Ours*	4,628	0	12.1	26.76
		[RA18]*	182	806	0.16	0.07
		[BBDC+11]*	182	2,684	0.16	0.07
2^{24}	11041	Ours	656	0	20.1	53.3
		[CLR17]	71	0	44.70	23.20
		[RA18]	342	48	0.71	0.67
		[BBDC+11]	342	160	0.71	0.67
	5535	Ours	806	0	22.01	25.09
		[CLR17]	64	0	40.10	20.10
		[RA18]	342	48	0.35	0.34
		[BBDC+11]	342	160	0.35	0.34
2^{20}	11041	Ours	43	0	4.49	20.49
		[CLR17]	6.4	0	6.40	11.50
		[RA18]	22	3	0.71	0.67
		[BBDC+11]	22	10	0.71	0.67
	5535	Ours	43	0	4.23	16.13
		[CLR17]	4.3	0	4.30	5.60
		[RA18]	22	3	0.35	0.34
		[BBDC+11]	22	10	0.35	0.34

Table 7.3: Our PSI protocol compared with [CLR17, RA18, BBDC+11] in the LAN setting for various set sizes. All executions are with a single thread with the exception (*) of $|X| = 2^{28}$, which is performed with 32 threads by Alice, and 4 threads by Bob. Communication/storage is in MB and running time is in seconds. The "Sender offline" column is running time required by Alice to initialize their database. It can be reused and is non-interactive.

length items.

7.7.3 Labeled PSI

We compare the performance of our Labeled PSI technique in the anonymous communication protocol Pung [AS16, ACLS17]. The Pung protocol allows a set of clients to privately send and retrieve messages through a server, without the server learning any information (including metadata) about the conversations. In each epoch of the protocol, a client wishes to privately retrieve several messages from the server from other clients using some secret keywords they share. This was achieved using a single-server PIR based on

—X—	—Y—	ℓ (Bytes)	Method	Server online	Client encrypt	Comm.
2^{20}	256	288	[ACLS17]	20.5s	4.92 s	120 MB
			Ours	4.6s	0.77s	17.6 MB

Table 7.4: Performance of Labeled PSI applied to the Pung anonymous communication protocol.

additive homomorphic encryption. In order for a client to obtain the index of messages sent to her, the server sends a Bloom filter containing the keyword-to-index mapping to each client. Pung also optimized for the multi-query using hashing techniques. In one setting, the client retrieves 256 messages in each epoch. Each message has 288 bytes, and a total number of 1 million messages is stored at the server. We used Labeled PSI to implement the retrieval process, and compared our performance to [ACLS17] in Figure 7.4. From the results, we see that Labeled PSI can achieve a 4.4× reduction in server's online computation time, and 6.8× reduction in communication.

Chapter 8

PSI From Private Information Retrieval

PIR-PSI: Scaling Private Contact discovery by Daniel Demmler, Ni Triue, Peter Rindal & Mike Rosulek, in PETS[DRRT18].

8.1 Introduction

With the widespread use of smartphones in the last decade, social networks and their connected instant messaging services are on the rise. Services like Facebook Messenger or WhatsApp connect more than a billion users worldwide.[1]

Contact discovery happens when a client initially joins a social network and intends to find out which of its existing contacts are also on this network. Even after this initial client join, it is also run periodically (*e.g.*, daily) in order to capture a client's contacts that join the network later on. A trivial approach for contact discovery is to send the client's entire address book to the service provider, who replies with the intersection of the client's contacts and the provider's customers. This obviously leaks sensitive client data to the service provider. In fact, a German court has recently ruled that such trivial contact discovery in the WhatsApp messaging service violates that country's privacy regulations [Pos17]. Specifically, a user cannot send her

[1] `https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day`

contact list to the WhatsApp servers for contact discovery, without written consent of all of her contacts.

A slightly better approach (often called "naïve hashing") has the client *hash* its contact list before sending it to the server. However, this solution is insecure, since it is prone to offline brute force attacks if the input domain is small (e.g., telephone numbers). Nonetheless, naïve hashing is used internally by Facebook and was previously used for contact discovery by the Signal messaging app. The significance of a truly private contact discovery was highlighted by the creators of Signal [Mar14].

8.1.1 State of the Art & Challenges

Contact discovery is fundamentally about identifying the intersection of two sets. There is a vast amount of literature on the problem of **private set intersection (PSI)**, in which parties compute the intersection of their sets without revealing anything else about the sets (except possibly their size). A complete survey is beyond the scope of this work, but we refer the reader to Pinkas et al. [PSZ18], who give a comprehensive comparison among the major protocol paradigms for PSI.

In contact discovery, the two parties have **sets of vastly different sizes**. The server may have 10s or 100s of millions of users in its input set, while a typical client has less than 1000.[2] However, most research on PSI is optimized for the case where two parties have sets of similar size. As a result, many PSI protocols have communication and computation costs that scale with the size of the larger set. For contact discovery, it is imperative that the client's effort (especially communication cost) scales *sublinearly* with the server's set size. Concretely, in a setting where the client is a mobile device, we aim for communication of at most a few megabytes. A small handful of works [CLR17, KLS$^+$17, RA18] focus on PSI for asymmetric set sizes. We give a comprehensive comparison of these works to ours in Section 2.2 and Section 8.7.

Even after solving the problems related to the client's effort, the computational cost to the server can also be prohibitive. For example, the server might have to perform expensive exponentiations for each item in its set. Unfortunately no known techniques allow the server to have computational

[2] A 2014 survey by Pew Research found that the average number of Facebook friends is 338 [Smi14].

cost sublinear in the size of its (large) input set. The best we can reasonably hope for (which we achieve) is for the server's computation to consist almost entirely of fast symmetric-key operations which have hardware support in modern processors.

If contact discovery were a one-time step only for new users of a service, then the difference between a few seconds in performance would not be a significant concern. Yet, *existing users* must also perform contact discovery to maintain an up-to-date view. Consider a service with 100 million users, each of which performs maintenance contact discovery once a week. This is only possible if the marginal cost of a contact discovery instance costs less than 6 milliseconds for the service provider (one week is roughly 600 million milliseconds)! To be truly realistic and practical, private contact discovery should be as fast as possible.

8.1.2 Chapter Contributions

We propose a new approach for private contact discovery that is practical for realistic set sizes. We refer to our paradigm as **PIR-PSI**, as it combines ideas from private information retrieval (PIR) and standard 2-party PSI.

Techniques: Importantly, we split the service provider's role into two to four non-colluding servers. With 2 servers, each one holds a copy of the user database. When a third server is used, 2 of the servers can hold secret shares of the user database rather than hold it in the clear. With 4 servers, all servers can hold secret shares. By using a computational PIR scheme, a single-server solution is possible. Most of our presentation focuses on our main contribution, the simpler 2-server version, but in Section 8.8 we discuss the other variants in detail. Note that multiple non-colluding servers is the traditional setting for PIR, and is what allows our approach to have sublinear cost (in the large server set size) for the client while still hiding its input set.

Roughly speaking, we combine highly efficient state-of-the-art techniques from 2-server PIR and 2-party private set intersection. The servers store their sets in a Cuckoo table so that a client with n items needs to probe only $O(n)$ positions of the server's database to compute the intersection. Using the state-of-the-art PIR scheme of Boyle, Gilboa & Ishai [BGI15, BGI16], each such query requires $O(\kappa \log N)$ bits of communication, where N is the size of the server's data. In standard PIR, the client learns the positions of the server's data in the clear. To protect the servers' privacy, we modify the

PIR scheme so that *one of the servers* learns the PIR output, but blinded by masks known to the client. This server and the client can then perform a standard 2-party PSI on masked values. For this we use the efficient PSI scheme of [KKRT16].

Within this general paradigm, we identify several protocol-level and systems-level optimizations that improve performance over a naïve implementation by several orders of magnitude. For example, the fact that the client probes *randomly distributed* positions in the server's database (a consequence of Cuckoo hashing with random hash functions) leads to an optimization that reduces cost by approximately 500×.

As a contribution of independent interest, we performed an extensive series of experiments (almost a trillion hashing instances) to develop a predictive formula for computing ideal parameters for Cuckoo hashing. This allows our protocol to use very tight hashing parameters, which can also yield similar improvements in all other cuckoo hashing based PSI protocols. A more detailed description can be found in Section 8.3.2.

Performance: Let n be the size of the client's set, let N the size of the server's set ($n \ll N$), and let κ be the computational security parameter. The total communication for contact discovery is $O\big(\kappa n \log(N \log n/\kappa n)\big)$. The computational cost for the client is $O\big(n \log(N \log n/\kappa n)\big)$ AES evaluations, $O(n)$ hash evaluations, and κ exponentiations. The exponentiations can be done once-and-for all in an initialization phase and re-used for subsequent contact discovery events between the same parties.

Each server performs $O\big((N \log n)/\kappa\big)$ AES evaluations, $O(n)$ hash evaluations, and κ (initialization-time) exponentiations. While this is indeed a large number of AES calls, hardware acceleration (i.e., AES-NI instructions and SIMD vectorization) can easily allow processing of a billion items per second per thread on typical server hardware. Furthermore, the server's computational effort in PIR-PSI is highly parallelizable, and we explore the effect of parallelization on our protocol.

Privacy/security for the client: If a corrupt server does not collude with the other server, then it learns nothing about the client's input set except its size. In the case where the two servers collude, even if they are malicious (*i.e.*, deviating arbitrarily from the protocol) they learn no more than the client's *hashed* input set. In other words, **the failure mode for PIR-PSI is to reveal no more than the naïve-hashing approach.** Since naïve-hashing is the status quo for contact discovery in practice, PIR-PSI is a strict privacy improvement.

Furthermore, the non-collusion guarantee is forward-secure. Compromising both servers leaks nothing about contact-discovery interactions that happened previously (when at most one server was compromised).

In PIR-PSI, malicious servers can use a different input set for each client instance (*e.g.*, pretend that Alice is in their database when performing contact discovery with Bob, but not with Carol). That is, the servers' effective data set is not anchored to some public root of trust (*e.g.*, a signature or hash of the "correct" data set).

Privacy/security for the server: A semi-honest client (*i.e.*, one that follows the protocol) learns no more than the intersection of its set with the servers' set, except its size. A malicious client can learn different information, but still no more than $O(n)$ bits of information about the servers' set (n is the purported set size of the client). We can characterize precisely what kinds of information a malicious client can learn.

Other features: PIR-PSI requires the servers to store their data-set in a fairly standard Cuckoo hashing table. Hence, the storage overhead is constant and updates take constant time.

PIR-PSI can be easily extended so that the client privately learns associated data for each item in the intersection. In the case of a secure messaging app, the server may hold a mapping of email addresses to public keys. A client may wish to obtain the public keys of any users in its own address book.

As mentioned previously, PIR-PSI can be extended to a 3-server or 4-server variant where some of the servers hold only *secret shares* of DB, with security holding if no two servers collude (cf. Section 8.8.2). This setting may be a better fit for practical deployments of contact discovery, since a service provider can recruit the help of other independent organizations, neither of which need to know the provider's user database. Holding the user database in secret-shared form reduces the amount of data that the service provider retains about its users[3] and gives stronger defense against data exfiltration.

[3] https://www.reuters.com/article/us-usa-cyber-signal/signal-messaging-app-turns-over-minimal-data-in-first-subpoena-idUSKCN1241JM

8.2 Preliminaries

Table 8.1: PIR-PSI parameters and symbols used.

Parameter	Symbol
symmetric security paramter [bits]	κ
statistical security paramter [bits]	λ
client set size [elements]	n
server set size [elements]	N
element length [bits]	ρ
cuckoo table size (Section 8.3.2)	m
DPF bins (Section 8.3.4)	β
DPF bin size [elements] (Section 8.3.4)	μ
PIR block size (Section 8.3.5)	b
scaling factor (Section 8.5.3)	c

8.2.1 Private Information Retrieval

Private Information Retrieval (PIR) was introduced in the 1990s by Chor et al. [CKGS98]. It enables a client to query information from one or multiple servers in a privacy preserving way, such that the servers are unable to infer which information the client requested. In contrast to the query, the servers' database can be public and may not need to be protected. When first thinking about PIR, a trivial solution is to have a server send the whole database to the client, who then locally performs his query. However, this is extremely inefficient for large databases. There exists a long list of works that improve PIR communication complexity [CMS99, GR05, GKL10, TP11, HOG11, DGH12, MBC13, HHG13, DC14, DHS14, AMBFK16, LG15, Hen16, BGI15, BGI16]. Most of these protocols demand multiple non-colluding servers. In this work, we are interested in 2-server PIR schemes.

8.2.2 Distributed Point Functions

Gilboa and Ishai [GI14] proposed the notion of a distributed point function (DPF). For our purposes, a DPF with domain size N consists of the following algorithms:

$\mathsf{DPF.Gen}$: a randomized algorithm that takes index $i \in [N]$ as input and outputs two (short) *keys* k_1, k_2.

$\mathsf{DPF.Expand}$: takes a short key k as input and outputs a long *expanded key* $K \in \{0,1\}^N$.[4]

The correctness property of a DPF is that, if $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$ then $\mathsf{DPF.Expand}(k_1) \oplus \mathsf{DPF.Expand}(k_2)$ is a string with all zeros except for a 1 in the ith bit.

A DPF's security property is that the marginal distribution of k_1 alone (resp. k_2 alone) leaks no information about i. More formally, the distribution of k_1 induced by $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$ is computationally indistinguishable from that induced by $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i')$, for all $i, i' \in [N]$.

PIR from DPF: Distributed point functions can be used for 2-party PIR in a natural way. Suppose the servers hold a database DB of N strings. The client wishes to read item $DB[i]$ without revealing i. Using a DPF with domain size N, the client can compute $(k_1, k_2) \leftarrow \mathsf{DPF.Gen}(i)$, and send one k_b to each server. Server 1 can expand k_1 as $K_1 = \mathsf{DPF.Expand}(k_1)$ and compute the inner product:

$$K_1 \cdot DB \overset{\text{def}}{=} \bigoplus_{j=1}^{N} K_1[j] DB[j]$$

Server 2 computes an analogous inner product. The client can then reconstruct as:

$$(K_1 \cdot DB) \oplus (K_2 \cdot DB) = (K_1 \oplus K_2) \cdot DB = DB[i]$$

since $K_1 \oplus K_2$ is zero everywhere except in position i.

BGI construction: Boyle, Gilboa & Ishai [BGI15, BGI16] describe an efficient DPF construction in which the size of the (unexpanded) keys is roughly $\kappa(\log N - \log \kappa)$ bits, where κ is the computational security parameter.

Their construction works by considering a full binary tree with N leaves. To expand the key, the $\mathsf{DPF.Expand}$ algorithm performs a PRF evaluation for each node in this tree. The (unexpanded) keys contain a PRF block for each level of the tree.

[4] The original DPF definition also requires efficient random access to this long expanded key. Our usage of DPF does not require this feature.

As described, this gives unexpanded keys that contain κ bits for each level of a tree of height $\log N$. To achieve $\kappa(\log N - \log \kappa)$ bits total, BGI suggest the following "early termination optimization": Treat the expanded key as a string of N/κ characters over the alphabet $\{0,1\}^\kappa$. This leads to a tree of height $\log(N/\kappa) = \log N - \log \kappa$. An extra κ-bit value is required to deal with the longer characters at the leaves, but overall the total size of the unexpanded keys is roughly $\kappa(\log N - \log \kappa)$ bits. In practice, we use hardware-accelerated AES-NI as the underlying PRF, with $\kappa = 128$.

8.3 Our Construction: PIR-PSI

We make use of the previously described techniques to achieve a practical solution for privacy-preserving contact discovery, called **PIR-PSI**. We assume that the service provider's large user database is held on 2 separate servers. To perform private contact discovery, a client interacts with both servers simultaneously. The protocol's best security properties hold when these two servers do not collude. Variants of our construction for 3 and 4 servers are described in Section 8.8.2, in which some of the servers hold only *secret shares* of the user database.

We develop the protocol step-by-step in the following sections. The full protocol can be found in Figure 8.1.

8.3.1 Warmup: PIR-PEQ

At the center of our construction is a technique for combining a *private equality test* (PEQ – a special case of PSI when the parties have one item each) [PSZ14] with a PIR query. Suppose a client holds private input i, x and wants to learn whether $DB[i] = x$, where the database DB is the private input of the servers.

First recall the PIR scheme from Section 8.2.1 based on DPFs. This PIR scheme has *linear reconstruction* in the following sense: the client's output $DB[i]$ is equal to the *XOR of the responses* from the two servers.

Suppose a PIR scheme with linear reconstruction is modified as follows: the client sends an additional mask r to server #1. Server #1 computes its PIR response v_1 and instead of sending it to the client, sends $v_1 \oplus r$ to server #2.

Then server #2 computes its PIR response v_2 and can reconstruct the masked result $v_2 \oplus (v_1 \oplus r) = DB[i] \oplus r$. We refer to this modification of PIR as **designated-output PIR**, as the client designates server #2 to learn the (masked) output.

The client can now perform a standard 2-party secure computation with server #2. In particular, they can perform a PEQ with input $x \oplus r$ from the client and $DB[i] \oplus r$ from the server. As long as the PEQ is secure, and the two servers do not collude, then the servers learn nothing about the client's input. If the two servers collude, they can learn i but not x.

This warm-up problem is not yet sufficient for computing private set intersection between a set X and DB, since the client may not know which location in DB to test against. Next we will address this by structuring the database as a Cuckoo hash table.

8.3.2 Cuckoo hashing

Cuckoo hashing has seen extensive use in Private Set Intersection protocols [PSZ14, PSSZ15, KKRT16, PSZ18, OOS17] and in related areas such as privacy preserving genomics [CCL+17]. This hashing technique uses an array of m bins and k hash functions $h_1, \ldots, h_k : \{0,1\}^* \to [m]$. The guarantee is that an item x will be stored in a hash table at one of the locations indexed by $h_1(x), ..., h_k(x)$. Furthermore, only a single item will be assigned to each bin. Typically k is quite small (we use $k = 3$). When inserting x into the hash table, a random index $i \in [k]$ is selected and x is inserted at location $h_i(x)$. If an item y currently occupies that location, it is evicted and y is re-inserted using the same technique. This process is repeated until all items are inserted or some upper bound on the number of trials have been reached. In that latter case, the procedure can either abort or place the item in a special location called the stash. We choose cuckoo hashing parameters such that this happens with sufficiently low probability (see Section 8.5.2); i.e., no stash is required.

In our setting the server encodes its set DB into a Cuckoo hash table. That way, the client (who has the much smaller set X) must probe only $k|X|$ positions of the Cuckoo table to compute the intersection. Using the PIR-PEQ technique just described makes the communication linear in $|X|$ but only logarithmic in $|DB|$.

8.3.3 Hiding the cuckoo locations

There is a subtle issue if one applies the PIR-PEQ idea naïvely. When the client learns that $y \in (DB \cap X)$, he/she will in fact learn whether y is placed in position $h_1(y)$ or $h_2(y)$ or $h_3(y)$ of the Cuckoo table. But this *leaks more than the intersection* $DB \cap X$, in the sense that it cannot be simulated given just the intersection! The placement of y in the Cuckoo table depends indirectly on all the items in DB.[5] Note that this is not a problem for other PSI protocols, since there the party who processes their input with Cuckoo hashing is the one who receives output from the PEQs. For contact discovery, we require these to be different parties.

To overcome this leakage, we design an efficient oblivious shuffling procedure that obscures the cuckoo location of an item. First, let us start with a simple case with two hash functions h_1, h_2, where the client holds a single item x. This generalizes in a natural way to $k = 3$ hash functions. Full details are provided in Section 8.10.

The client will generate and send two PIR queries, for positions $h_1(x)$ and $h_2(x)$ of DB. The client also sends two masks r_1 and r_2 to server #1 which serve as masks for the designated-output PIR. Server #1 randomly chooses whether to swap these two masks. That is, it chooses a random permutation $\sigma : \{1, 2\} \to \{1, 2\}$ and masks the first PIR query with $r_{\sigma(1)}$ and the second with $r_{\sigma(2)}$. Server #2 then reconstructs the designated PIR output, obtaining $DB[h_1(x)] \oplus r_{\sigma(1)}, DB[h_2(x)] \oplus r_{\sigma(2)}$. The client now knows that if $x \in DB$, then server #2 must hold either $x \oplus r_1$ or $x \oplus r_2$.

Now instead of performing a private equality test, the client and server #2 can perform a standard **2-party PSI** with inputs $\{x \oplus r_1, x \oplus r_2\}$ from the client and the designated PIR values $\{DB[h_1(x)] \oplus r_{\sigma(1)}, DB[h_2(x)] \oplus r_{\sigma(2)}\}$ from server #2. This technique perfectly hides whether x was found at $h_1(x)$ or $h_2(x)$. While it is possible to perform a separate 2-item PSI for each PIR query, it is actually more efficient (when using the 2-party PSI protocol of [KKRT16]) to combine all of the PIR queries into a single PSI with $2n$ elements each.

[5] For instance, say the client holds set X and (somehow) knows the server has set $DB = X \cup \{z\}$ for some unknown z. It happens that for many $x \in X$ and $i \in [k]$, $h_i(x)$ equals some location ℓ. Then with good probability some $x \in X$ will occupy location ℓ. However, after testing location ℓ the client learns no $x \in X$ occupies this location. Then the client has learned some information about z (namely, that $h_i(z) = \ell$ is likely for some $i \in [k]$), even though z is not in the intersection.

Because of the random masks, this approach may introduce a false positive, where $DB[j] \neq x$ but $DB[j] \oplus r = x \oplus r'$ (for some masks r and r'), leading to a PSI match. In our implementation we only consider items of length 128, so the false positive probability taken over all client items is only $2^{-128+\log_2((2n)^2)}$, by a standard union bound.

8.3.4 Optimization: Binning queries

The client probes the servers' database in positions that are determined by the Cuckoo hash functions. Under the reasonable assumptions that (1) the client's input items are chosen independently of the Cuckoo hash functions and (2) the cuckoo hash functions are random functions, the client probes DB in *uniformly distributed* positions.

Knowing that the client's queries are randomly distributed in the database, we can take advantage of the fact that the queries are "well-spread-out" in some sense. Consider dividing the server database (N entries) into β bins of equal size. The client will query the database in nk random positions, so the distribution of these queries into the β bins can be modeled as a standard balls and bins problem. We can choose a number β of bins and a maximum load μ so that Pr[there exists a bin with $\geq \mu$ balls] is below some threshold (say, 2^{-40} in practice). With such parameters, the protocol can be optimized as follows.

The parties agree to divide DB into β regions of equal size. The client computes the positions of DB that he/she wishes to query, and collects them according to their region. The client adds dummy PIR queries until there are *exactly* μ queries to each region. The dummy items are necessary because revealing the number of (non-dummy) queries to each region would leak information about the client's input to the server. For each region, the server treats the relevant N/β items as a sub-database, and the client makes exactly μ PIR queries to that sub-database.

This change leads to the client making more PIR queries than before (because of the dummy queries), but each query is made to a much smaller PIR instance. Looking at specific parameters shows that binning can give **significant performance improvements.**

It is well-known that with $\beta = O\big(nk/\log(nk)\big)$ bins, the maximum number of balls in any bin is $\mu = O\big(\log(nk)\big)$ with very high probability. The total number of PIR queries (including dummy ones) is $\beta\mu = \Theta(nk)$. That is, the binning optimization with these parameters increases the number of PIR queries by a constant factor. At the same time, the PIR database instances are all smaller by a large factor of $\beta = O\big(nk/\log(nk)\big)$. The main bottleneck in PIR-PSI is the computational cost of the server in answering PIR queries, which scales linearly with the size of the PIR database. Reducing the size of all effective PIR databases by a factor of β has a significant impact on performance. In general, tuning the constant factors in β (and corresponding μ) gives a wide trade-off between communication and computation.

8.3.5 Optimization: Larger PIR Blocks

So far we have assumed a one-to-one correspondence between the entries in the server's cuckoo table and the server's database for purposes of PIR. That is, we invoke PIR with an v-item database corresponding to a region of the cuckoo table with v entries.

Suppose instead that we use PIR for an $v/2$-item database, where each item in the PIR database consists of a block of 2 cuckoo table entries. The client generates each PIR query for a single item, but now the PIR query returns a block of 2 cuckoo table entries. The server will feed both entries into the 2-party PSI, so that these extra neighboring items are not leaked to the client.

This change affects the various costs in the following ways: (1) It reduces the number of cryptographic operations needed for the server to answer each PIR query by half; (2) It does not affect the computational cost of the final inner product between the expanded DPF key and PIR database entries, since this is over the same amount of data; (3) It reduces the communication cost of each PIR query by a small amount (κ bits); (4) It doubles all costs of the 2-party PSI, since the server's PSI input size is doubled.

Of course, this approach can be generalized to use a PIR blocks of size b, so that a PIR database of size v/b is used for v cuckoo table entries. This presents a trade-off between communication and computation, discussed further in in Section 8.5.

8.3.6 Asymptotic Performance

With these optimizations the computational complexity for the client is the generation of $\beta\mu = O(n)$ PIR queries of size $O(\log(N/\kappa\beta))$. As such they perform $O(n\log(N/\kappa\beta)) = O(n\log(N\log n/\kappa n))$ calls to a PRF and send $O(\kappa n\log(N/(\kappa n\log n)))$ bits. The servers must expand each of these queries to a length of $O(N/\beta)$ bits which requires $O(N\mu/\kappa) = O(N\log n/\kappa)$ calls to a PRF.

8.4 Security

8.4.1 Semi-Honest Security

The most basic and preferred setting for PIR-PSI is when at most one of the parties is passively corrupt (a.k.a. semi-honest). This means that the corrupt party does not deviate from the protocol. Note that restricting to a single corrupt party means that we assume *non-collusion* between the two PIR servers.

Theorem 19. *The $\mathcal{F}_{\mathsf{pir-psi}}$ protocol (Figure 8.1) is a realization of $\mathcal{F}^{n,N}_{\mathsf{psi}}$ secure against a semi-honest adversary that corrupts at most one party in the $\mathcal{F}^{nk,\beta\mu}_{\mathsf{psi}}$ hybrid model.*

Proof. In the semi-honest non-colluding setting it is sufficient to show that the transcript of each party can be simulated given their input and output. That is, we consider three cases where each one of the parties is individually corrupt.

Corrupt Client: Consider a corrupt client with input X and output $Z = X \cap DB$. We show a simulator that can simulate the view of the client given just X and Z. First observe that the simulator playing the role of both servers knows the permutation $\pi = \pi_2 \circ \pi_1$ and the vector of masks r. As such, response v can be computed as follows. For $x \in Z$ the simulator randomly samples one of k masks $r_{i_1}, \ldots, r_{i_k} \in r$ which the client will use to mask x and add $x \oplus r_{i_j}$ to v. Pad v with random values not contained in union u to size $\beta\mu$ and forward v to the ideal $\mathcal{F}^{nk,\beta\mu}_{\mathsf{psi}}$. Conditioned on no spurious collisions between v and u in the real interaction (which happen with negligible probability, following the discussion in Section 8.3.3) this ideal interaction perfectly simulates the real interaction.

One additional piece of information learned by the client is that cuckoo hashing on the set DB with hash function $h_1, ..., h_k$ succeeded. However, by the choice of cuckoo parameter, this happens with overwhelming probability and therefore the simulator can ignore the case of cuckoo hashing failure.

Corrupt server: Each server's view consists of:

- PIR queries (DPF keys) from the client; since a single DPF key leaks nothing about the client's query index, these can be simulated as dummy DPF keys.
- Messages in the oblivious masking step, which are uniformly distributed as discussed in Section 8.3.3 and Appendix 8.10.
- In the case of server #2, masked PIR responses from server #1, which are uniformly distributed since they are masked by the $\vec{r}$ values. □

8.4.2 Colluding Servers

If the two servers collude, they will learn both DPF keys for every PIR query, and hence learn the locations of all client's queries into the cuckoo table. These locations indeed leak information about the client's set, although the exact utility of this leakage is hard to characterize. The servers still learn nothing from the PSI subprotocol by colluding since only one of the servers is involved.

It is worth providing some context for private contact discovery. The state-of-the-art for contact discovery is a naïve (insecure) hashing protocol, where both parties simply hash each item of their set, the client sends its hashed set to the server, who then computes the intersection. This protocol is insecure because the server can perform a dictionary attack on the client's inputs.

However, *any PSI protocol* (including ours) can be used in the following way. First, the parties hash all their items, and then use the hashed values as inputs to the PSI. As long as the hash function does not introduce collisions, pre-hashing the inputs preserves the correctness of the PSI protocol.

A side effect of pre-hashing inputs is that the parties never use their "true" inputs to the PSI protocol. Therefore, the PSI protocol cannot leak more than the hashed inputs — identical to what the status quo naïve hashing protocol leaks. Again, this observation is true for *any* PSI protocol. In the specific case of PIR-PSI, if parties pre-hash their inputs, then even if the two servers collude (even if they are malicious), the overall protocol can never leak more about the client's inputs than naïve hashing. Relative to existing solutions implemented in current applications, that use naïve hashing, there is no extra security risk for the client to use PIR-PSI.

8.4.3 Malicious Client

Service providers may be concerned about malicious behavior (i.e., protocol deviation) by clients during contact discovery. Since servers get no output from PIR-PSI, there is no concern over a malicious client inducing inconsistent output for the servers. The only concern is therefore what unauthorized information a malicious client can learn about DB.

Overall the only information the client receives in PIR-PSI is from the PSI subprotocol. We first observe that the PSI subprotocol we use ([KKRT16]) is naturally secure against a malicious client, when it is instantiated with an appropriate OT extension protocol. This fact has been observed in [Lam16, OOS17]. Hence, in the presence of a malicious client we can treat the PSI subprotocol as an ideal PSI functionality. The malicious client can provide at most nk inputs to the PSI protocol — the functionality of PSI implies that the client therefore learns no more than nk bits of information about DB. This leakage is comparable to what an honest client would learn by having an input set of nk items.

Modifications for more precise leakage characterization: In DPF-based PIR schemes clients can make malformed PIR queries to the server, by sending k_1, k_2 so that $\mathsf{DPF.Expand}(k_1) \oplus \mathsf{DPF.Expand}(k_2)$ has more than one bit set to 1. The result of such a query will be the XOR of several DB positions.

However, Boyle et al. [BGI16] describe a method by which the servers can ensure that the client's PIR queries (DPF shares) are well-formed. The technique increases the cost to the servers by a factor of roughly 3× (but adds no cost to the client).

The client may also send malformed values in the oblivious masking phase. But since the servers use those values independently of DB, a simulator (who sees the client's oblivious masking messages to both servers) can simulate what masks will be applied to the PIR queries. Overall, if the servers ensure validity of the client's PIR values, we know that server #1's input to PSI will consist of a collection of nk individual positions from DB, each masked with values that can be simulated.

8.5 Implementation

We implemented a prototype of our $\mathcal{F}_{\mathsf{pir-psi}}$ protocol described in Figure 8.1. Our implementation uses AES as the underlying PRF (for the distributed point function of [BGI16], and relies on the PSI implementation of [KKRT16] and the oblivious transfer from [Rin17]. Upon publication, our implementation will be made publicly available.

8.5.1 System-level Optimizations

We highlight here system-level optimizations that contribute to the high performance of our implementation. We analyze their impact on performance in Section 8.12.

Optimized DPF full-domain evaluation: Recall that the DPF construction of [BGI16] can be thought of as a large binary tree of PRF evaluations. Expanding the short DPF key corresponds to computing this entire tree in order to learn the values at the leaves. The process of computing the values of all the leaves is called "full-domain evaluation" in [BGI16].

DPF full-domain evaluation is the major computational overhead for the servers in our protocol. To limit its impact our implementation takes full advantage of instruction vectorization (SIMD). Most modern processors are capable of performing the same instruction on multiple (*e.g.*, 8) pieces of data. However, to fully utilize this feature, special care has to be taken to ensure that the data being operated on is in cache and contiguous.

To meet these requirements, our implementation first evaluates the top 3 levels of the DPF binary tree, resulting in 8 remaining independent subtrees.

We then perform SIMD vectorization to traverse all 8 subtrees simultaneously. Combining this technique with others, such as the removal of `if` statements in favor of array indexing, our final implementation is roughly 20× faster then a straight-forward (but not careless) sequential implementation and can perform full-domain DPF evaluation at a rate of 2.6 billion bits/s on a single core.

Single-pass processing: With the high raw throughput of DPF evaluation, it may not be surprising that it was no longer the main performance bottleneck. Instead, performing many passes over the dataset (once for each PIR query) became the primary bottleneck by an order of magnitude. To address this issue we further modify the workflow to evaluate all DPFs (PIR queries) for a single bin in parallel using vectorization.

That is, for all μ DPF evaluations in a given bin, we evaluate the binary trees in parallel, and traverse the leaves in parallel. The values at the leaves are used to take an inner product with the database items, and the parallel traversal ensures that a given database item only needs to be loaded from main memory (or disk) once. This improves (up to 5×) the performance of the PIR protocol on large datasets, compared to the straightforward approach of performing multiple sequential passes of the dataset.

Parallelization: Beyond the optimizations listed above, we observe that our protocol simply is very amenable to parallelization. In particular, our algorithm can be parallelized both within the DPF evaluation using different subtrees and by distributing the PIR protocols for different bins between several cores/machines. In the setting where thousands of these protocols are being executed a day on a fixed dataset, distributing bin evaluations between different machines can be extremely attractive due to the fact that several protocol instances can be batched together to gain even greater benefits of vectorization and data locality. The degree of parallelism that our protocol allows can be contrasted with more traditional PSI protocols which require several global operations, such as a single large shuffle of the server's encoded set (as in [PSZ14, PSSZ15, KKRT16, PSZ18, OOS17]).

8.5.2 Cuckoo Hashing Parameters

To achieve optimal performance it is crucial to minimize the size of the cuckoo table and the number of hash functions. The table size affects how much work the servers have to perform and the number of hash functions k

affects the number of client queries. As such we wish to minimize both parameters while ensuring cuckoo hashing failures are unlikely as this leaks a single bit about DB. Several works [FMM09, DGM$^+$10] have analyzed cuckoo hashing from an asymptotic perspective and show that the failure probability decreases exponentially with increasing table size. However, the exact relationship between the number of hash functions, stash size, table size and security parameter is unclear from such an analysis.

We solve this problem by providing an accurate relationship between these parameters through extensive experimental analysis of failure probabilities. That is, we ran Cuckoo-hashing instances totalling nearly 1 trillion items hashed, over two weeks for a variety of parameters. As a result our bounds are significantly more accurate and general than previous experiments [PSZ18, CLR17]. We analyzed the resulting distribution to derive highly predictive equations for the failure probability. We find that $k = 2$ and $k \geq 3$ behave significantly different and therefore derive separate equations for each.

Our extrapolations are graphed in Figure 8.2 & 8.3, and the specifics of the formulas are given in Section 8.11.

8.5.3 Parameter Selection for Cuckoo Hashing & Binning

Traditional use of cuckoo hashing instructs the parties to sample new hash functions for each protocol invocation. In our setting however it can make sense to instruct the servers, which hold a somewhat static dataset, to perform cuckoo hashing once and for all. Updates to the dataset can then be handled by periodically rebuilding the cuckoo table once it has reached capacity. This leaves the question of what the cuckoo hashing success probability should be. It is standard practice to set statistical failure events like this to happen with probability 2^{-40}. However, since the servers perform cuckoo hashing only occasionally (and since hashing failure applies only to initialization, not future queries), we choose to use more efficient parameters with a security level of $\lambda = 20$ bits, i.e., $\Pr[\text{Cuckoo failure}] = 2^{-20}$. We emphasize that once the items are successfully placed into the hash table, all future *lookups* (e.g., contact discovery instances) are error-free, unlike, say, in a Bloom filter.

We also must choose the number of hash functions to use.[6] Through exper-

[6] Although the oblivious shuffling procedure of Section 8.3.3 can be extended in a natural

imental testing we find that overall the protocol performs best with $k = 3$ hash functions. The parameters used can be computed by solving for $e \approx 1.4$ in Equation 8.2 given that $\lambda = 20$.

To see why this configuration was chosen we must also consider another important parameter, the number of bins β. Due to binning being performed for each protocol invocation by the client, we must ensure that it succeeds with very high probability and therefore we use the standard of $\lambda = 40$ to choose binning parameters. An asymptotic analysis shows that the best configuration is to use $\beta = O(n/\log n)$ bins, each of size $\mu = O(\log n)$. However, this hides the exact constants which give optimal performance. Upon further investigation we find that the choice of these parameters result in a communication/computation trade-off.

For the free variable c, we set the number of bins to be $\beta = cn/\log_2 n$ and solve for the required bin size μ. As can be seen in Figure 8.4, the use of $k = 3$ and scaling factor $c = 4$ result the best running time at the expense of a relatively high communication of 11 MiB. However, at the other end of the spectrum is $k = 2$ and $c = 1/16$ results in the smallest communication of 2.4 MiB. The reason $k = 2$ achieves smaller communication for any fixed c is that the client sends $k = 2$ PIR queries per item instead of three. However, $k = 2$ requires that the cuckoo table is three times larger than for $k = 3$ and therefore the computation is slower.

Varying c affects the number of bins β. Having fewer bins reduces the communication due to the bins being larger and thereby having better real/dummy query ratio. However, larger bins also increases the overall work, since the work is proportional to the bin size μ times N. We aim to minimize both the communication and running time. We therefore decided on choosing $k = 3$ and $c = 1/4$ as our default configuration, the circled data-point in Figure 8.4. However, we note that in specific settings it may be desirable to adjust c further.

The PIR block size b also results in a computation/communication trade-off. Having a large block size gives shorter PIR keys and therefore less work to expand the DPF. However, this also results the server having a larger input set to the subsequent PSI which makes that phase require more communication and computation. Due to the complicated nature of how all these parameters interact with each other, we empirically optimized the parameters to find that a block size between 1 and 32 gives the best trade-off.

way to include a stash, we use a stash-free variant of Cuckoo hashing.

8.6 Performance

In this section we analyze the performance of PIR-PSI. We ran all experiments on a single benchmark machine which has 2x 18-core Intel Xeon E5-2699 2.30 GHz CPU and 256 GB RAM. Specifically, we ran all parties on the same machine, communicating via localhost network, and simulated a network connection using the Linux `tc` command: a LAN setting with 0.02 ms round-trip latency, 10 Gbps network bandwidth; a WAN setting with a simulated 80 ms round-trip latency, 100 Mbps network bandwidth. We process elements of size $\rho = 128$ bits.

Our protocol can be separated into two phases: the *server's init* phase when database is stored in the Cuckoo table, and the *contact discovery* phase where client and server perform the private intersection.

8.6.1 PIR-PSI performance Results

In our *contact discovery* phase, the client and server first perform the preprocessing of PSI between n and $3n$ items which is independent of parties' inputs. We refer this step as preprocessing phase which specifically includes base OTs, and $O(n)$ PRFs. The online phase consists of protocol steps that depend on the parties' inputs. To understand the scalability of our protocol, we evaluate it on the range of server set size $N \in \{2^{20}, 2^{24}, 2^{26}, 2^{28}\}$ and client set size $n \in \{1, 4, 2^8, 2^{10}\}$. The small values of $n \in \{1, 4\}$ simulate the performance of incremental updates to a client's set. Table 8.6.1 presents the communication cost and total *contact discovery* time with online time for both single- and multi-threaded execution with $T \in \{1, 4, 16\}$ threads.

As discussed in Section 8.5.3, there are a communication and computation trade-off on choosing the different value c and b which effects the number of bins and how many items are selected per PIR query. The interplay between these two variable is somewhat complex and offer a variety of communication computation trade-offs. For smaller $n \in \{1, 4\}$, we set $b = 32$ to drastically reduce the cost of the PIR computation at the cost of larger PSI. For larger n, we consider parameters which optimize running time and communication separately, and show both in Table 8.6.1.

Our experiments show that our PIR-PSI is highly scalable. For the database size $N = 2^{28}$ and client set size $n = 2^{10}$, we obtain an overall running time of 33.02 s and only 4.93 MiB bits of communications for the contact discovery

Param.				Comm.	Running time [seconds]		
N	n	c	b	[MiB]	$T = 1$	$T = 4$	$T = 16$
2^{28}	2^{10}	4	16	28.3	4.07	1.60	0.81
		0.25	1	4.93	33.02	13.22	5.54
	2^{8}	3	16	7.10	3.61	1.30	0.65
		1	1	2.20	14.81	6.92	3.40
	4	1	32	0.06	1.93	–	–
	1	1	32	0.03	1.21	–	–
2^{26}	2^{10}	2	8	12.7	1.61	0.72	0.41
		0.25	1	4.28	7.22	3.65	1.36
	2^{8}	6	16	10.3	0.98	0.51	0.26
		0.25	4	1.36	4.36	1.90	0.97
	4	1	32	0.06	0.56	–	–
	1	1	32	0.03	0.48	–	–
2^{24}	2^{10}	1	8	8.61	0.67	0.36	0.22
		0.25	1	3.85	2.28	0.94	0.50
	2^{8}	4	8	4.81	0.49	0.22	0.18
		1	1	1.68	1.26	0.57	0.36
	4	1	32	0.05	0.19	–	–
	1	1	32	0.03	0.16	–	–
2^{20}	2^{10}	0.5	4	2.10	0.22	0.10	0.06
		0.25	1	2.98	0.32	0.21	0.16
	2^{8}	2	4	1.95	0.20	0.09	0.06
		0.25	4	1.13	0.24	0.18	0.15
	4	1	32	0.05	0.14	–	–
	1	1	32	0.03	0.13	–	–

Table 8.2: PIR-PSI protocol's total contact discovery communication cost and running time using T threads, and $\beta = cn/\log_2 n$ bins and PIR block size of b. LAN: 10 Gbps, 0.02 ms latency.

phase using a single thread. Alternatively, running time can be reduced to just 4 s for the cost of 28 MiB communication. Increasing the number of threads from 1 to 16, our protocol shows a factor of 5× improvement, due to the fact that it parallelizes well. When considering the smallest server set size of $N = 2^{20}$ with 16 threads, our protocol requires only 1.1 MiB of communication and 0.24 s of contact discovery time.

We point out that the computational workload for the client is small and consists only of DPF key generation, sampling random values and the classical PSI protocol in the size of the client set. This corresponds to 10% of the overall running time (e.g., 0.3 s of the total 3.6 s for $N = 2^{28}$ and $n = 2^8$). Despite our experiments being run on a somewhat powerful server, the overwhelming bottleneck for performance is the computational cost for the server. Hence, our reported performance is representative of a scenario in which the client is a weaker device (e.g., a mobile device).

Detailed numbers on the performance impact of the optimizations from Section 8.5.1 are provided in Section 8.12.

8.6.2 Updating the Client and Server Sets

In addition to performing PIR-PSI on sets of size n and N, the contact discovery application requires periodically adding items to these sets. In case the client adds a single item x to their set X, only the new item x needs to be compared with the servers' set. Our protocol can naturally handle this case by simply having the client use $X' = \{x\}$ as their input to the protocol. However, a shortcoming of this approach is that we cannot use binning and the PIR query spans the whole cuckoo table.

For a database of size $N = 2^{24}$, our protocol requires only 0.16 s and 0.19 s to update 1 and 4 items, respectively. When increasing the size to $N = 2^{28}$, we need 1.9 s to update one item. Our update queries are cheap in terms of communication, roughly 30–50 kiB. We remark that update queries can be parallelized well due to the fact that DPF.Gen and DPF.Expand can each be processed in a divide and conquer manner. Also, several update queries from different users can be batched together to offer very high throughput. However, our current implementation only supports parallelization at the level of bins/regions, and not for a single DPF query.

The case when a new item is added to the servers' set can easily be handled by performing a traditional PSI between one of the servers and the client, where the server only inputs their new item. One could also consider batching several server updates together, and then performing a larger PSI or applying our protocol to the batched server set.

8.7 Comparison with Prior Work

In this section we give a thorough qualitative & quantitative comparison between our protocol and those of CLR [CLR17], KLSAP [KLS+17], and RA [RA18]. We obtained the implementations of CLR & KLSAP from the respective authors, but the implementation of RA is not publicly available. Because of that, we performed a comparison on inputs of size $N \in \{2^{16}, 2^{20}, 2^{24}\}$ and $n \in \{5535, 11041\}$ to match the parameters used in [RA18, Table 1&2]. While the experiments of RA were performed on an Intel Haswell i7-4770K

Protocol	Parameters		Communication	Running time [seconds]				Client	Server Init.	
	N	n	Size [MiB]	LAN (10 Gbps)		WAN (100 Mbps)		Storage	[seconds]	
				$T = 1$	$T = 4$	$T = 1$	$T = 4$	[MiB]	$T = 1$	$T = 4$
CLR [CLR17]	2^{24}	11041	**21.1**	38.6	19.7	41.0	22.1	**0.00**	76.8	20.6
		5535	**12.5**	34.0	16.3	36.0	18.2		71.2	18.5
	2^{20}	11041	11.5	3.7	3.2	4.9	4.4		9.1	2.5
		5535	5.6	3.5	1.9	4.1	2.5		5.1	1.4
	2^{16}	11041	4.1/4.4	1.8	1.4	2.2	1.8		1.2	0.3
		5535	2.6	0.9	0.6	1.1	0.9		0.9	0.3
KLSAP [KLS+17]	2^{24}	11041	2049 (43.3)	90.4	–	265.1	–	1941	8.32	–
		5535	1070 (21.7)	52.3	–	128.3	–	1016		
	2^{20}	11041	1968 (43.3)	82.1	–	259.9	–	1860	0.58	–
		5535	989 (21.7)	44.8	–	124.7	–	935		
	2^{16}	11041	1963 (43.3)	81.8	–	259.6	–	1855	0.04	–
		5535	984 (21.7)	44.0	–	121.4	–	930		
RA [RA18]	2^{24}	11041	171.67 (0.67)*	**1.08***	–	18.39*	–	171.00*	333.62	–
		5535	168.34 (0.34)*	**0.75***	–	17.61*	–	168.00*		
	2^{20}	11041	**11.36 (0.67)***	0.67*	–	**3.41***	–	10.69*	20.78	–
		5535	**10.84 (0.34)***	0.34*	–	2.89*	–	10.50*		
	2^{16}	11041	**1.34 (0.67)***	0.66*	–	**1.33***	–	0.67*	1.30	–
		5535	**1.00 (0.34)***	0.33*	–	**0.85***	–	0.66*		
Ours	2^{24}	11041	32.46	2.18	**1.65**	**5.63**	**5.13**	**0.00**	**2.690**	–
		5535	21.45	1.34	**1.11**	**3.72**	**2.77**			
	2^{20}	11041	22.86	**0.37**	**0.31**	3.70	**3.59**		**0.089**	–
		5535	11.67	**0.29**	**0.24**	**2.50**	**2.29**			
	2^{16}	11041	12.83	**0.28**	**0.29**	2.55	**2.55**		**0.004**	–
		5535	7.66	**0.21**	**0.20**	1.85	**1.85**			

Table 8.3: Comparison of PIR-PSI to CLR, KLSAP, and RA with $T \in \{1, 4\}$ threads. LAN: 10 Gbps, 0.02 ms latency. WAN: 100 Mbps, 80 ms latency. Best results marked in bold. Online communication reported in parenthesizes. Cells with "-" denote the setting is not supported. Cells with "*" indicate that the numbers are scaled for a fair comparison of error probability. CLR and RA use 32 bit items, while PIR-PSI and KLSAP process 128 bit items.

quadcore CPU with 3.4 GHz and 16 GB RAM, we ran the KLSAP and CLR protocols on our own hardware, described in the previous section. We remark that RA's benchmark machine has 3.4 GHz, which is 1.48× *faster than our machine*. The number of cores and RAM available on our hardware does not influence the results of the single-threaded benchmarks ($T = 1$). Results of the comparison are summarized in Table 8.3.

8.7.1 CLR protocol

The high level idea of the protocol of Chen, Laine & Rindal (CLR) [CLR17] is to have the client encrypt each element in their dataset under a homomorphic encryption scheme, and send the result to the server. The server homomorphically evaluates the intersection circuit on encrypted data, and sends back the result for the receiver to decrypt.

The CLR protocol has communication complexity $O(\rho n \log(N))$, where the items are ρ bits long. Ours has communication complexity $O(\kappa n \log(N/(\kappa n \log n)))$, with no dependence on ρ since the underlying PSI protocol [KKRT16] has no such dependence. For small items (*e.g.*, $\rho = 32$ as reflected in Table 8.3), CLR uses less communication than our protocol, *e.g.*, 20 MiB as opposed to 37 MiB. However, their protocol scales very poorly for string length of 128 bits as it would require significantly less efficient FHE parameters. Furthermore, CLR can not take advantage of the fact that most contact lists have significantly fewer than 5535 entries. That is, the cost for $n = 1$ and $n = 5535$ is roughly equivalent, because of the way FHE optimizations like batching are used. The main computational bottleneck in CLR is the server performing $O(n)$ homomorphic evaluations on large circuits of size $O(N/n)$. The comparable bottleneck in our protocol is performing $\mathsf{DPF.Expand}$ and computing the large inner products. Since these operations take advantage of hardware-accelerated AES, PIR-PSI is significantly faster than CLR, *e.g.*, 20× for $N = 2^{24}$.

The server's initialization in CLR involves hashing the N items into an appropriate data structure (as in PIR-PSI), but also involves pre-computing the many coefficients of polynomials. Hence our initialization phase is much faster than CLR, *e.g.*, 40× for $N = 2^{24}$.

The CLR protocol does not provide a full analysis of security against malicious clients. Like our protocol, the leakage allowed with a malicious client is likely to be minimal.

8.7.2 KLSAP protocol

In the KLSAP [KLS+17] protocol, the server sends a Bloom filter of size $O(\lambda N)$ to the client in an offline phase. During later contact discovery phases, the client refers to this Bloom filter.

The size of this Bloom filter is considerable: nearly 2 GiB for $N = 2^{24}$ server items. This data, which must be stored by the client, may be prohibitively large for mobile client applications. By contrast, our protocol (and CLR) requires no long-term storage by the client.

In the contact discovery phase, KLSAP runs Yao's protocol to obliviously evaluate an AES circuit for each of the client's items. Not even counting the boom filter, this requires slightly more communication than our approach (1.5×). Additionally, it requires more computation by the (weak) client: evaluating many AES garbled circuits (thousands of AES calls per item) vs. running many instances $\mathsf{DPF.Gen}$ ($\log N$ AES calls per item) followed by a specialized PSI protocol (constant number of hash/AES per item). Even though the server in our protocol must perform $O(N)$ computation during contact discovery, our considerable optimizations result in a much faster discovery phase (40× for $N = 2^{24}$).

When the server makes changes to its set in KLSAP, it must either re-key its AES function (which results in re-sending the huge Bloom filter), or send incremental updates to the Bloom filter (which breaks forward secrecy, as a client can query its items in both the old and new versions of the Bloom filter).

KLSAP is easily adapted to secure against a malicious client. This stems from the fact that the contact discovery phase uses Yao's protocol with the client acting as garbled circuit evaluator. Hence it is naturally secure against a malicious client (provided that one uses malicious-secure OTs).

Subtleties about hashing errors: The way that KLSAP uses a Bloom filter also leads to qualitative differences in the error probabilities compared to PIR-PSI. In KLSAP the server publishes a Bloom filter for all clients, who later query it for membership. The false-positive rate (FPR) of the Bloom filter is the probability that a single item not in the server's set is mistakenly classified as being in the intersection. Importantly, the FPR for this global Bloom filter is *per client item.* In KLSAP this FPR is set to 2^{-30}, which means after processing a combined 1 million client items the probability of some client receiving a false positive may be as high as 2^{-10}!

By contrast, the PIR-PSI server places its items in a Cuckoo table once-and-for all (with hashing error probability 2^{-20}). As long as this *one-time event* is successful, all subsequent probes to this data structure are error-free (we store the entire item in the Cuckoo table, not just a short fingerprint as in [RA18]). If the hashing is unsuccessful, the server simply tries again with different hash functions. All of our other failure events (*e.g.*, probability of a bad event within our 2-party PSI protocol) are calibrated for error probability 2^{-40} *per contact discovery instance*, not per item! To have a comparable guarantee, the Bloom filter FPR of KLSAP would have to be scaled by a factor of $\log_2(n)$.

8.7.3 RA protocol

The RA [RA18] protocol uses a similar approach to KLSAP, in that it uses a relatively large representation of the server's set, which is sent in an offline phase and stored by the client. The downsides of this architecture discussed above for KLSAP also apply to RA (client storage, more client computation, false-positive rate issues, forward secrecy).

RA's implementation uses a Cuckoo filter that stores for each item a 16-bit fingerprint. This choice leads to a relatively high false-positive rate of $2^{-13.4}$. To achieve the failure events with error probability 2^{-40} per contact discovery instance (in line with our protocol), the Cuckoo filter FPR of RA would be $2^{-(40+\log_2(n))}$. Therefore, their protocol would have to be modified to use 56-bit and 57-bit fingerprints for $n = 5535$ and $n = 11041$, respectively. This change increases the communication cost, transmission time, and offline storage requirements $3.44 - 3.5\times$, relative to the numbers reported in [RA18, Table 1]. In Table 8.3 we report the *scaled* communication costs, the *scaled* online running time, and the *scaled* client's storage, but refrain from trying to scale the server's initialization times. As can be seen our protocol running time is $1.2 - 3.2\times$ faster than RA for sufficiently large N. We also have a 100× more efficient server initialization phase and achieve communication complexity of $O(n \log N)$ as compared to $O(N)$ of RA. This difference can easily be seen by how the communication of RA significantly increases for larger N.

In RA, the persistent client storage is not a Bloom filter but a more compact Cuckoo filter. This reduces the client storage, but it still remains linear in N. For $N = 2^{28}$ the storage requirement is 2.57 GiB to achieve an error probability of 2^{-40} per contact discovery instance.

The RA protocol does not provide any analysis of security against malicious clients.

8.8 Other Extensions

Although this work mainly focuses on the setting of pure contact discovery with two servers, our protocol can be easily modified for other settings.

8.8.1 PSI with associated data (PSI+AD)

refers to a scenario where the client has a set A of keys and the server has a set B of key-value pairs, and the client wishes to learn $\{(k, v) \mid (k, v) \in B \text{ and } k \in A\}$. In the context of an encrypted messaging service, the keys may be phone numbers or email addresses, and the values may be the user's public key within the service.

PIR-PSI can be modified to support associated data, in a natural way. The server's Cuckoo hash table simply holds key-value pairs, and the 2-party PSI protocol is replaced by a 2-party PSI+AD protocol. The client will then learn *masked* values for each item in the intersection, which it can unmask. The PSI protocol of [KKRT16] that we use is easily modified to allow associated data.

8.8.2 3- and 4-Server Variant

We described PIR-PSI in the context of two non-colluding servers, who store identical copies of the service provider's user database. Since both servers hold copies of this sensitive database, they are presumably both operated by the service provider, so the promise of non-collusion may be questionable. Using a folklore observation from the PIR literature, we can allow servers to hold only *secret shares* of the user database, at the cost of adding more servers.

Consider the case of 3 servers. The service provider can recruit two independent entities to assist with private contact discovery, without entrusting them with the sensitive user database. The main idea is to let servers #2 and #3 hold additive secret shares of the database and jointly simulate the effect of a single server that holds the database in the clear.

Recall the 2-party DPF-PIR scheme of [BGI16], that we use. The client sends DPF shares k_1, k_2 to the servers, who expand the keys to K_1, K_2 and performs an inner product with the database. The client XORs the two responses to obtain result $(K_1 \cdot DB) \oplus (K_2 \cdot DB) = (K_1 \oplus K_2) \cdot DB = DB[i]$.

In our 3-server case, we have server #1 holding DB, and servers #2 & #3 holding DB_2, DB_3 respectively, where $DB = DB_2 \oplus DB_3$. We simply let the client send DPF share k_1 to server #1, and send k_2 to *both* of the other servers. All servers expand their DPF share and perform an inner product with their database/share. The client will receive $K_1 \cdot DB$ from server #1, $K_2 \cdot DB_2$ from server #2, and $K_2 \cdot DB_3$ from server #3. The XOR of all responses is indeed

$$\begin{aligned} &(K_1 \cdot DB) \oplus (K_2 \cdot DB_2) \oplus (K_2 \cdot DB_3) \\ &= (K_1 \cdot DB) \oplus K_2 \cdot (DB_2 \oplus DB_3) \\ &= K_1 \cdot DB \oplus K_2 \cdot DB = (K_1 \oplus K_2) \cdot DB = DB[i] \end{aligned}$$

Now the entire PIR-PSI protocol can be implemented with this 3-server PIR protocol as its basis. The computational cost of each server is identical to the 2-server PIR-PSI, and is performed in parallel by the independent servers. Hence, the total time is minimally effected. The client's total communication is unaffected since server #2 can forward K_2 to server #3. The protocol security is the same, except that the non-collusion properties hold now only if server #1 doesn't collude with any of the other servers. If servers #2 & #3 collude, then they clearly learn DB, but as far as the client's privacy is concerned, the situation simply collapses to 2-server PIR-PSI.

Similarly, server #1 can also be replaced by a pair of servers, each with secret shares (and this sharing of DB can be independent of the other sharing of DB). This results in a 4-server architecture with security for the client as long as neither of servers #1 & #2 collude with one of the servers #3 & #4, and where no single server holds DB in the clear.

8.8.3 2-Server with OPRF Variant

An alternative to the 3-server variant above is to leverage a pre-processing phase. Similar to [RA18], the idea is to have server #1 apply an oblivious PRF to their items instead of a hash function, which will ensure that the database is pseudorandom in the view of server #2, who does not know the PRF key. In particular, let server #1 sample a key k for the oblivious PRF F used in [RA18] and update the database as $DB'_i := F_k(DB_i)$, which is then sent to server #2. When a client wishes to compute the intersection of its set X with DB, they first perform an oblivious PRF protocol with server #1 to learn $X' = \{F_k(x) \mid x \in X\}$. Note that this protocol ensures that the client does not learn k. The client can now engage in our standard two-server PIR-PSI protocol to compute $Z' = X' \cap DB'$ and thereby infer $Z = X \cap DB$.

The advantage of this approach is that server #2 does not learn any information about the plaintext database DB since the PRF was applied to each record. Moreover, this holds even if server #2 colludes with one of the clients. The added performance cost of this variant has two components. First, server #1 must update their database by applying the PRF to it. As shown by [RA18], a single CPU core can process roughly 50,000 records per seconds, which is sufficiently fast given that this is a one-time cost. The second overhead is performing the oblivious PRF protocol with the clients. This requires three exponentiations per item in X, which represents an acceptable overhead given that $|X|$ is small.

8.8.4 Single Server Variant

We also note that our PIR-PSI architecture has the potential to be extended to the single-server setting. Several PIR protocols [AS16, ACLS17, AMBFK16] based on fully homomorphic encryption have been shown to offer good performance while at the same time removing the two-server requirement. With some modifications to our architecture, we observe that such PIR protocols can be used. The main challenge to overcome is how to mask and shuffle the results of the PIR before being forwarded to the PSI protocol. Instead of simply returning the resulting PIR values of [AS16, ACLS17, AMBFK16], the server can return a secret share of it. These shares can then be obliviously shuffled and forwarded to a PSI protocol as done by our PIR-PSI architecture. We leave the optimization and exact specification of such a single-server PIR-PSI protocol to future work, but note its feasibility.

8.9 Deployment

We now turn our attention to practical questions surrounding the real-world deployment of our multi-server PIR-PSI protocol. As briefly discussed in the previous section, the requirement that a single organization has two non-colluding servers may be hard to realize. However, we argue that the 3-server or 2-server with an OPRF variants make deployment significantly simpler. Effectively, these variants reduce the problem to finding one or two external semi-honest parties that will not collude with the service provider (server #1). A natural solution to this problem is to leverage existing cloud providers such as Amazon Web Services and/or Microsoft Azure. Given that these companies have a very large incentive to maintain their reputation, they would have a large interest to not collude. Alternatively, other well regarded privacy conscious organization such as the Electronic Frontier Foundation could serve as the second server.

8.10 Hiding Cuckoo Locations

We previously described a simple approach to hide the Cuckoo location for a single item of the client. At first glance, it seems trivial to generalize this approach to many items for the client – simply repeat the procedure above once for each client item. However, it requires server #2 to know that two PIR queries correspond to the same logical client item (*e.g.*, two queries correspond to $h_1(x)$ and $h_2(x)$ for the same x, so their masks can be randomly swapped). This turns out to be incompatible with another of our optimizations (see Section 8.3.4) that lets the servers learn some information about the location of the PIR queries. It is safe to leak this information about the *collective* set of client queries (*e.g.*, a certain number of the client's queries are made to this region in DB) but not about *specific* queries (*e.g.*, client has an x where $h_1(x)$ is in this region and $h_2(x)$ is in that region).

We therefore generalize this oblivious masking technique as the following functionality:

- The client holds a permutation π that maps its logical inputs to the indices of those PIR queries. That is, for each item x_j of the client, the $\pi(2j+1)$'th PIR query is for $DB[h_1(x_j)]$ and the $\pi(2j+2)$'th PIR query is for $DB[h_2(x_j)]$.
- The client also holds a vector $\vec{r}$ of masks
- Server #1 chooses a random permutation σ with the property that $\{\sigma(2j+1), \sigma(2j+2)\} = \{2j+1, 2j+2\}$ for all j. That is, σ consists of swaps of adjacent items only.
- Servers #1 & #2 have vectors of PIR responses $\vec{v}_1, \vec{v}_2$, respectively.

- The goal is for server #2 to learn:

$$\vec{v}_1 \oplus \vec{v}_2 \oplus \pi(\sigma(\vec{r})) \overset{\text{def}}{=} \vec{v}_1 \oplus \vec{v}_2 \oplus \big(r_{\pi(\sigma(1))}, \ldots, r_{\pi(\sigma(2n))}\big)$$

We claim that this results in masks r_{2j+1}, r_{2j+2} being "routed" to the two PIR queries corresponding to logical item x_j. Indeed, since $\vec{v}_1 \oplus \vec{v}_2$ comprise the *unmasked* PIR outputs, the definition of π implies that

$$\begin{aligned} \vec{v}_1 \oplus \vec{v}_2 = \pi\big(&DB[h_1(x_1)], DB[h_2(x_1)], \ldots, \\ &DB[h_1(x_n)], DB[h_2(x_n)]\big) \end{aligned}$$

Hence, server #2's output is the following vector permuted by π:

$$\big(DB[h_1(x_1)], DB[h_2(x_1)], \ldots\big) \oplus (r_{\sigma(1)}, r_{\sigma(2)}, \ldots)$$

By the construction of σ, we see that masks r_{2j+1} and r_{2j+2} are indeed paired up with $DB[h_1(x_j)]$ and $DB[h_2(x_j)]$, as desired.

Hence, the client can compute the $2n$ values of the form $x_j \oplus r_{2j+1}, x_j \oplus r_{2j+2}$, and use these as input to a conventional 2-party PSI protocol. Server #2 can use its output from this oblivious masking as its input to the PSI. From the output of this PSI subprotocol, the client can deduce the intersection.

To actually achieve this oblivious masking functionality, we do the following: The client picks three random mask vectors $\vec{r}, \vec{s}, \vec{t}$ of length m and generates a 2-out-of-2 secret sharing of π as $\pi = \pi_2 \circ \pi_1$. The client sends $\vec{t}$ and π_2 to server #2 and $\vec{r}$, $\vec{s}$, π_1 and $\vec{t} \oplus \pi(\vec{s})$ to server #1. Server #1 sends $\pi_1\big(\sigma(\vec{r}) \oplus \vec{s}\big)$ and $[\vec{t} \oplus \pi(\vec{s}) \oplus \vec{v}_1]$ to server #2, who can then compute $\vec{v}$:

$$\begin{aligned} \vec{v} &= \vec{v}_2 \oplus \pi_2\Big(\pi_1\big(\sigma(\vec{r}) \oplus \vec{s}\big)\Big) \oplus \vec{t} \oplus [\vec{t} \oplus \pi(\vec{s}) \oplus \vec{v}_1] \\ &= \vec{v}_2 \oplus \pi\big(\sigma(\vec{r})\big) \oplus \pi(\vec{s}) \quad \oplus \vec{t} \oplus \ \vec{t} \oplus \pi(\vec{s}) \oplus \vec{v}_1 \\ &= \vec{v}_1 \oplus \vec{v}_2 \oplus \pi\big(\sigma(\vec{r})\big) \end{aligned}$$

In order to be compatible with further optimizations, we must show that the servers learn nothing about the client's permutation π, which captures which PIR queries correspond to the same logical client input.

Server #1 receives $\vec{r}, \vec{s}, \pi_1$, and $\vec{t} \oplus \pi(\vec{s})$. These values are randomly selected by the client, so server #1 learns nothing about π from this oblivious masking process.

Server #2 receives $\vec{t}$ and π_2 from the client, and $\pi_1(\sigma(\vec{r}) \oplus \vec{s})$ as well as $\vec{t} \oplus \pi(\vec{s}) \oplus \vec{v}_1$ from server #1. Since the values $\vec{t}, \pi_2, \vec{r}, \vec{s}$ are each uniformly distributed, the entire view of server #2 is random. Hence, server #2 likewise learns nothing about π from the oblivious masking process.

Saving bandwidth: We can save bandwidth in the oblivious masking procedure by observing that many of the client's messages are random, and can instead be chosen pseudorandomly.

Recall that $\vec{r}, \vec{s}, \pi_1, \vec{t} \oplus \pi(\vec{s})$ are sent to server #1 and $\vec{t}, \pi_2$ to server #2. The client can send a small seed w to server #1 and use this seed to pseudorandomly choose $(\vec{r}, \vec{s}, \pi_1) = \mathrm{PRG}(w)$. Similarly, the client can send a seed to server #2 and use it pseudorandomly define $\vec{t}$.

Now that π_1 is fixed, the client can solve for appropriate π_2 such that $\pi_2 \circ \pi_1 = \pi$. The client must send other values explicitly: $\vec{t} \oplus \pi(\vec{s})$ to server #1 and π_2 to server #2.

8.11 Cuckoo Hashing Failure Probability Formula

Let $e > 1$ be the expansion factor denoting that N items are inserted into a cuckoo table of size eN. Figure 8.2 shows the security parameter (i.e., λ, such that the probability of hashing failure is $2^{-\lambda}$) of Cuckoo hashing with $k = 2$ hash functions. As N becomes larger, λ scales linearly with $\log_2 N$ and with the stash size s, which matches the results of [DGM+10]. For $e \geq 8$ and $k = 2$, we interpolate the relationship as the linear equation

$$\lambda = \big(1 + 0.65s\big)\big(3.3\log_2(e) + \log_2(N) - 0.8\big) \tag{8.1}$$

For smaller values of e, we observe that λ quickly converges to 1 at $e = 2$. We approximate this behavior by subtracting $\big(5\log_2(N) + 14\big)e^{-2.5}$ from Equation 8.1. We note that these exact interpolated parameters are specific to our implementation which uses a specific eviction policy (linear walk) and re-insert bounds (100). Moreover, we only consider the case of $\lambda \geq 1$. However, we observed similar parameters for other variations.

We also consider the case $k = 3$, shown in Figure 8.3 and find that it scales significantly better that $k = 2$. For instance, at $e = 2$ we find $\lambda \approx 100$ for interesting set sizes while the same value of e applied to $k = 2$ results in

$\lambda \approx 1$. As before we find that λ grows linearly with the expansion factor e. Unlike in the case of $k = 2$, we observe that increasing N has a slight negative effect on λ. Namely, doubling N roughly decreases λ by 2. However, the slope at which λ increases for $k = 3$ is much larger than $k = 2$ and therefore this dependence on $\log N$ has little impact on λ. We summarize these findings for $k = 3$ as the linear equation

$$\lambda = a_N e + b_N \tag{8.2}$$

where $a_N \approx 123.5$ and $b_N \approx -130 - \log_2 N$. Here we use an approximation to hide an effect that happens for small $N \leq 512$. In this regime we find that the security level quickly falls. In particular, the slope a_N and intercept b_N go to zero roughly following the normal distribution CDF. By individually interpolating these variable we obtain accurate predictions of λ for $N \geq 4$. Our interpolations show that $a_N = 123.5 \cdot \text{CDF}_{\text{NORMAL}}(x = N, \mu = 6.3, \sigma = 2.3)$ and $b_N = -130 \cdot \text{CDF}_{\text{NORMAL}}(x = N, \mu = 6.45, \sigma = 2.18) - \log_2 N$.

For $k = 3$ we do not consider a stash due to our experiments showing it having a much smaller impact as compared to $k = 2$. Additionally, we do not compute exact parameters for $k > 3$ due to the diminishing returns. In particular, $k = 4$ follows the same regime as $k = 3$ but only marginally improves the failure probability.

8.12 Effect of the Optimizations

In this section, we discuss the effect of our optimizations on the performance. By far the most important optimization employed is the use of *binning*. Observe in Table 8.12 that the running time with all optimizations enabled is 1.0 s while the removal of binning results in a running time of 1906 s. This can be explained by the overall reduction of asymptotic complexity to $O(N \log n)$ with binning as opposed to $O(Nn)$ without binning.

Another important optimization is the use of PIR blocks which consist of more than cuckoo table item. This *blocking* technique allows for a better balance between the cost of the PIR compared to the cost of the subsequent PSI. Increasing the block size logarithmically decreases the cost of the PIR while linearly increasing the cost of the PSI. Since the PIR computation is so much larger than the PSI (assuming $n \ll N$) setting the block size to be greater than 1 gave significant performance improvements. In practice we

N	n	All Opt. Enabled	No Batching	No Blocking	No Vectorization	No Binning
2^{24}	2^{12}	1.0	2.1	3.7	40.1	1906

Table 8.4: Online running time in seconds of the protocol with all optimizations enabled compared with the various optimizations of Section 8.5.1 individually disabled.

found that setting b to be within 1 and 32 gave the best results. Table 8.12 shows that setting b to optimize running time gives a 3.7× improvement.

We also consider the effect that our highly optimized DPF implementation has on the overall running time. *Vectorization* refers to an implementation of the DPF with the full-domain optimization implemented similar as described by the [BGI16, Figure 4]. We then improve on their basic construction to take full advantage of CPU vectorization and fixed-key AES. The result is a 40× difference in overall running-time.

The final optimization is to improve memory locality of our implementation by carefully accessing the cuckoo table. Instead of computing each PIR query individually, which would require loading the large cuckoo table from memory many times, our *batching* optimization runs all DPF evaluations for a given database location at the same time. This significantly reduces the amount of data that has to be fetched from main memory. For a dataset of size $N = 2^{24}$ we observe that this optimization yields 2.1× improvement, and an even bigger 5× improvement when applied to a larger dataset of $N = 2^{28}$ along with using $T = 16$ threads.

Parameters: X is the client's set, DB is the set held by server #1 and #2, where $X, DB \subseteq \{0,1\}^\rho$. $n = |X|, N = |DB|$, k is the number of cuckoo hash functions. The protocol uses an instance of $\mathcal{F}_{\mathsf{psi}}$ with input length ρ. λ, κ are the statistical and computational security parameter.

1. **[Cuckoo Hashing]** The servers agree on k random hash function $h_1, ..., h_k : \{0,1\}^\rho \to [m]$ and cuckoo table size $m = |H|$ such that inserting N items into cuckoo hash table H succeeds with probability $\geq 1 - 2^{-\lambda}$.

   The servers compute a cuckoo hash table H such that for all $y \in DB$, $H[h_i(y)] = y$ for some $i \in k$.

2. **[Query]** Upon the client receiving their set X,

   (a) Send $n = |X|$ to the servers. All parties agree on the number of bins $\beta = O(n/\log n)$, and their size $\mu = O(\log n)$ (see Section 8.5.3). Define the region DB_i as all locations $j \in [m]$ of H such that $(i-1)\frac{m}{\beta} < j \leq i\frac{m}{\beta}$.

   (b) For $x \in X, h \in \{h_1, ..., h_k\}$, let $h(x)$ index the j'th location in bin i. The client adds (x, j) to bin $B[i]$.

   (c) For bin $B[i]$,

      i. Pad $B[i]$ to size μ with the pair $(\perp, 0)$.

      ii. For $(x, j) \in B[i]$ in a random order, the client constructs the keys $k_1, k_2 = \mathsf{DPF.Gen}(j)$. Send k_s to server $\#s$.

      iii. Server $\#s$ expands their key $K_s = \mathsf{DPF.Expand}(k_s)$ and compute $v_s[\ell] = DB_i \cdot K_s$ where k_s is the ℓ'th DPF key received.

3. **[Shuffle]** Observe that, $(v_1 \oplus v_2)[\ell] = H[j_\ell]$ where j_ℓ is the ℓ'th PIR location.

   (a) The client samples a permutation π of $\beta\mu$ items such that for the i'th $x \in X$, and $j \in [k]$ it holds that the $\pi\big((i-1)k + j\big)$'th DPF key corresponded to the query of item x at location $h_j(x)$.

   (b) The client samples $w_1, w_2 \leftarrow \{0,1\}^\kappa$ and sends w_1 to server #1, and w_2 to server #2. Define shared terms $t = \textsc{prg}(w_2)$, $(r||s||\pi_1) = \textsc{prg}(w_1)$ where r, s, t are random vectors of the same size as v_i and π_1 is a random permutation of $\beta\mu$ items. The client sends π_2 to server #2 such that $\pi_2 \circ \pi_1 = \pi$ and sends $e = t \oplus \pi(s)$ to server #1.

   (c) Server #1 samples a random permutation σ of $\beta\mu$ items such that for all $i \in [\beta\mu/k]$ it holds that $\sigma(j) \in S_i$ where $j \in S_i = (i-1)k + \{1, ..., k\}$. Server #1 sends $p_1 = \pi_1\big(\sigma(r) \oplus s\big)$ and $p_2 = e \oplus v_1$ to server #2.

   (d) Server #2 computes $v = v_2 \oplus \pi_2(p_1) \oplus t \oplus p_2$.

4. **[PSI]** The client then computes the masked versions of the $x_i \in X$ as $x'_i = \{x_i \oplus r_{\pi\big((i-1)k+1\big)}, ..., x_i \oplus r_{\pi(ik)}\}$ and computes u as the union of all these sets. The client and server #2 respectively send u, v to $\mathcal{F}_{\mathsf{psi}}^{nk,\beta\mu}$ such that the client receives $z = u \cap v$. The client outputs $\{x_i : x'_i \cap z \neq \emptyset\}$.

Figure 8.1: Our PIR-PSI protocol $\mathcal{F}_{\mathsf{pir-psi}}$.

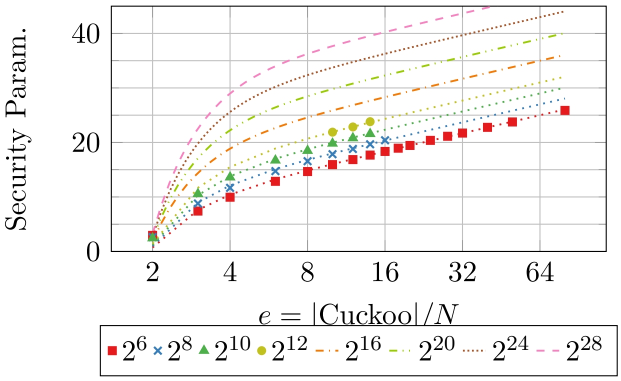

Figure 8.2: Empirical and interpolated (dashed/dotted lines) cuckoo success probability for $k = 2$ hash functions. Series are for different set sizes N and labels as $\log_2 N$. 28p and 24p are extrapolated bounds for set sizes $N = 2^{24}, 2^{28}$ respectively.

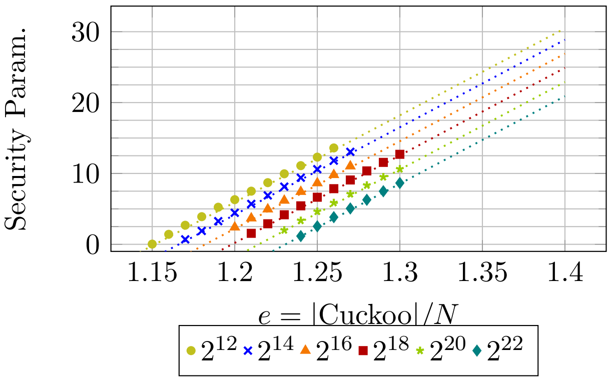

Figure 8.3: Empirical and interpolated (dashed/dotted lines) cuckoo success probability for $k = 3$ hash functions. Series are for different set sizes N and labels as $\log_2 N$.

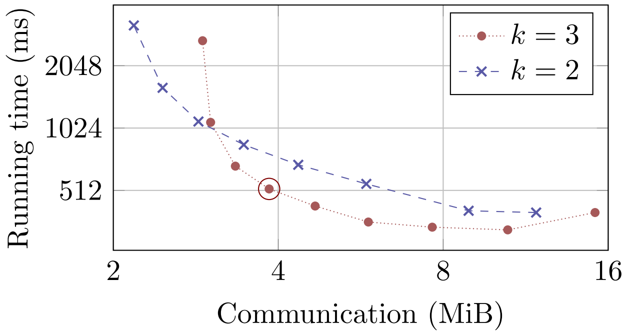

Figure 8.4: Communication and computation trade-off for $n = 2^{10}, N = 2^{24}, T = 16$ threads, k cuckoo hash function, no stash, with the use of $\beta = cn/\log_2 n$ bins where $c \in \{2^{-5}, 2^{-4}, \ldots, 2^{3}\}$ and are listed left to right as seen above. The configuration $(k = 2, c = 2^{-5})$ did not fit on the plot. The highlighted point $(k = 3, c = 1/4)$ is the default parameter choice that is used.

Bibliography

[ABC+15] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. Technical report, IACR Cryptology ePrint Archive (2015/1192), 2015.

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007.

[ACLS17] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. Cryptology ePrint Archive, Report 2017/1142, 2017. https://eprint.iacr.org/2017/1142.

[AJLA+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 483–501. Springer, 2012.

[Alb17] Martin R Albrecht. On dual lattice attacks against small-secret lwe and parameter choices in helib and seal. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 103–129. Springer, 2017.

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 535–548. ACM Press, November 2013.

[ALSZ15] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 673–701. Springer, Heidelberg, April 2015.

[AMBFK16] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir : Private information retrieval for everyone. *PoPETs*, 2016(2):155 – 174, 2016.

[ANS10a] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard Cuckoo hashing: Constant worst-case operations with a succinct representation. In *51st FOCS*, pages 787–796. IEEE Computer Society Press, October 2010.

[ANS10b] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Foundations of Computer Science (FOCS), 2010 51st Annual IEEE Symposium on*, pages 787–796. IEEE, 2010.

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[AS16] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569, 2016.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[BBDC$^+$11] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering gattaca: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 691–702. ACM, 2011.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *28th ACM STOC*, pages 479–488. ACM Press, May 1996.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016.

[BFT16] Tatiana Bradley, Sky Faber, and Gene Tsudik. Bounded size-hiding private set intersection. In Vassilis Zikas and Roberto De Prisco, editors, *SCN 16*, volume 9841 of *LNCS*, pages 449–467. Springer, Heidelberg, August / September 2016.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public-Key Cryptography–PKC 2013*, pages 1–13. Springer, 2013.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 1292–1303. ACM Press, October 2016.

[BGKS13] Raef Bassily, Adam Groce, Jonathan Katz, and Adam Smith. Coupled-worlds privacy: Exploiting adversarial uncertainty in statistical data privacy. In *54th FOCS*, pages 439–448. IEEE Computer Society Press, October 2013.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[BNO08] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In David Wagner, editor, *CRYPTO 2008*, volume 5157 of *LNCS*, pages 451–468. Springer, Heidelberg, August 2008.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Heidelberg, August 2012.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. *Network*, 1(101101), 2010.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CCD$^+$17] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[CCL$^+$17] Gizem S. Cetin, Hao Chen, Kim Laine, Kristin E. Lauter, Peter Rindal, and Yuhou Xia. Private queries on encrypted genomic data. *IACR Cryptology ePrint Archive*, 2017:207, 2017.

[CCMS17] T-H. Hubert Chan, Kai-Min Chung, Bruce Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. Cryptology ePrint Archive, Report 2017/1033, 2017. http://eprint.iacr.org/2017/1033.

[CGBL$^+$17] Melissa Chase, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, and Peter Rindal. Private collaborative neural network learning. Cryptology ePrint Archive, Report 2017/762, 2017. https://eprint.iacr.org/2017/762.

[CGGI16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–33. Springer, 2016.

[CGN97] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords.* Citeseer, 1997.

[CGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *CANS 12*, volume 7712 of *LNCS*, pages 218–231. Springer, Heidelberg, December 2012.

[CJJ$^+$13] David Cash, Stanislaw Jarecki, Charanjit S. Jutla, Hugo Krawczyk, Marcel-Catalin Rosu, and Michael Steiner. Highly-scalable searchable symmetric encryption with support for Boolean queries. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 353–373. Springer, Heidelberg, August 2013.

[CJS14] Ran Canetti, Abhishek Jain, and Alessandra Scafuro. Practical UC security with a global random oracle. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 14*, pages 597–608. ACM Press, November 2014.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[CKL15] Jung Hee Cheon, Miran Kim, and Kristin Lauter. Homomorphic computation of edit distance. In *International Conference on Financial Cryptography and Data Security*, pages 194–212. Springer, 2015.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1243–1255. ACM Press, October / November 2017.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. Technical report, Cryptology ePrint Archive, Report 2018/105, 2018.

[CS16] Ana Costache and Nigel P Smart. Which ring based somewhat homomorphic encryption scheme is best? In *Cryptographers' Track at the RSA Conference*, pages 325–340. Springer, 2016.

[DC14] Changyu Dong and Liqun Chen. A fast single server private information retrieval protocol with low communication cost. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part I*, volume 8712 of *LNCS*, pages 380–399. Springer, Heidelberg, September 2014.

[DCKT10] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. *Linear-Complexity Private Set Intersection Protocols Secure in Malicious Model*, pages 213–231. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 13*, pages 789–800. ACM Press, November 2013.

[DGBL$^+$15] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. Technical report, Microsoft Research, 2015. http://research.microsoft.com/apps/pubs/default.aspx?id=258435.

[DGH12] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. Cryptology ePrint Archive, Report 2012/083, 2012. http://eprint.iacr.org/2012/083.

[DGM$^+$10] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *International Colloquium on Automata, Languages, and Programming*, pages 213–225. Springer, 2010.

[DHS14] Daniel Demmler, Amir Herzberg, and Thomas Schneider. RAID-PIR: Practical multi-server PIR. In *ACM Workshop on Cloud Computing Security*, CCSW '14, pages 45–56, New York, NY, USA, 2014. ACM.

[DM03] Luc Devroye and Pat Morin. Cuckoo hashing: further analysis. *Information Processing Letters*, 86(4):215–219, 2003.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 265–284. Springer, Heidelberg, March 2006.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. Cryptology ePrint Archive, Report 2018/579, 2018. https://eprint.iacr.org/2018/579.

[DS16] Léo Ducas and Damien Stehlé. Sanitization of FHE ciphertexts. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–310. Springer, 2016.

[Eri16] Ericsson. Ericsson mobility report: On the pulse of the networked society. *Stockholm, Sweden*, 2016.

[FIPR05] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

[FJNT16] Tore Kasper Frederiksen, Thomas P. Jakobsen, Jesper Buus Nielsen, and Roberto Trifiletti. On the complexity of additively homomorphic UC commitments. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part I*, volume 9562 of *LNCS*, pages 542–565. Springer, Heidelberg, January 2016.

[FMJS17] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. Distributed measurement with private set-union cardinality. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 2295–2312. ACM Press, October / November 2017.

[FMM09] Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 490–503. Springer, 2009.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In Christian Cachin and Jan Camenisch, editors, *EUROCRYPT 2004*, volume 3027 of *LNCS*, pages 1–19. Springer, Heidelberg, May 2004.

[FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 271–282. Springer, 2003.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. `http://eprint.iacr.org/`.

[GBDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 201–210, 2016.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Advances in cryptology–crypto 2012*, pages 850–867. Springer, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology–CRYPTO 2012*, pages 850–867. Springer, 2012.

[GHV10] Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. i-hop homomorphic encryption and rerandomizable yao circuits. In *Annual Cryptology Conference*, pages 155–172. Springer, 2010.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[GKL10] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In Phong Q. Nguyen and David Pointcheval, editors, *PKC 2010*, volume 6056 of *LNCS*, pages 107–123. Springer, Heidelberg, May 2010.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *49th Annual Allerton Conference on Communication, Control, and Computing, Allerton 2011, Allerton Park & Retreat Center, Monticello, IL, USA, 28-30 September, 2011*, pages 792–799, 2011.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 803–815. Springer, Heidelberg, July 2005.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO (1)*, pages 75–92, 2013.

[Ham] Shane Hamlin. Electronic registration information center. http://www.ericstates.org/.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[Hen16] Ryan Henry. Polynomial batch codes for efficient IT-PIR. Cryptology ePrint Archive, Report 2016/598, 2016. http://eprint.iacr.org/2016/598.

[HFH99] Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *In Proc. of the 1st ACM Conference on Electronic Commerce*, pages 78–86. ACM Press, 1999.

[HHG13] Ryan Henry, Yizhou Huang, and Ian Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *NDSS 2013*. The Internet Society, February 2013.

[HJP13] W. Hart, F. Johansson, and S. Pancratz. FLINT: Fast Library for Number Theory, 2013. Version 2.4.0, http://flintlib.org.

[HKE12] Yan Huang, Jonathan Katz, and David Evans. Quid-Pro-Quo-tocols: Strengthening semi-honest protocols with dual execution. In *2012 IEEE Symposium on Security and Privacy*, pages 272–284. IEEE Computer Society Press, May 2012.

[HL10] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. *Journal of Cryptology*, 23(3):422–456, July 2010.

[HMRT12] Carmit Hazay, Gert Læssøe Mikkelsen, Tal Rabin, and Tomas Toft. Efficient RSA key generation and threshold Paillier in the two-party setting. In Orr Dunkelman, editor, *CT-RSA 2012*, volume 7178 of *LNCS*, pages 313–331. Springer, Heidelberg, February / March 2012.

[HOG11] Ryan Henry, Femi G. Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 11*, pages 677–690. ACM Press, October 2011.

[HS14] Shai Halevi and Victor Shoup. Algorithms in helib. In *International cryptology conference*, pages 554–571. Springer, 2014.

[IKN$^+$17a] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[IKN$^+$17b] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. http://eprint.iacr.org/2017/738.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[JL10] Stanisław Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *International Conference on Security and Cryptography for Networks*, pages 418–435. Springer, 2010.

[KKNO17] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Accessing data while preserving privacy. *CoRR*, abs/1706.01552, 2017.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 16*, pages 818–829. ACM Press, October 2016.

[KL16] Rachel Player Kim Laine, Hao Chen. Simple encrypted arithmetic library - SEAL (v2.1). Technical report, September 2016.

[KLS$^+$17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4):177–197, 2017.

[KMRR15] Vladimir Kolesnikov, Payman Mohassel, Ben Riva, and Mike Rosulek. Richer efficiency/security trade-offs in 2PC. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part I*, volume 9014 of *LNCS*, pages 229–259. Springer, Heidelberg, March 2015.

[KMRS14] Seny Kamara, Payman Mohassel, Mariana Raykova, and Seyed Saeed Sadeghian. Scaling private set intersection to billion-element sets. In Nicolas Christin and Reihaneh Safavi-Naini, editors, *FC 2014*, volume 8437 of *LNCS*, pages 195–215. Springer, Heidelberg, March 2014.

[KOS15a] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure ot extension with optimal overhead. Cryptology ePrint Archive, Report 2015/546, 2015. https://eprint.iacr.org/2015/546.

[KOS15b] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 724–741. Springer, Heidelberg, August 2015.

[KOV14] Peter Kairouz, Sewoong Oh, and Pramod Viswanath. Differentially private multi-party computation: Optimality of non-interactive randomized response. *arXiv preprint arXiv:1407.1546*, 2014.

[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, August 2005.

[KS08] Shiva Prasad Kasiviswanathan and Adam Smith. A note on differential privacy: Defining resistance to arbitrary side information. Cryptology ePrint Archive, Report 2008/144, 2008. http://eprint.iacr.org/2008/144.

[KSK$^+$18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. Cryptology ePrint Archive, Report 2018/254, 2018. https://eprint.iacr.org/2018/254.

[Lam16] Mikkel Lamaek. Breaking and fixing private set intersection protocols. Cryptology ePrint Archive, Report 2016/665, 2016. https://eprint.iacr.org/2016/665.

[Lar17] Large-Scale Data & Systems (LSDS) Group, Imperial College, London. spectre-attack-sgx. Github Repository, 2017. https://github.com/lsds/spectre-attack-sgx.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg, January 2015.

[Lin16] Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. https://eprint.iacr.org/2016/046.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, pages 1–23, 2010.

[Mar14] Moxie Marlinspike. The difficulty of private contact discovery. A company sponsored blog post, 2014. https://whispersystems.org/blog/contact-discovery/.

[Mar17] Moxie Marlinspike. Technology preview: Private contact discovery for signal. Signal blog post, 2017. https://signal.org/blog/private-contact-discovery/.

[MBC13] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. PIRMAP: Efficient private information retrieval for MapReduce. In Ahmad-Reza Sadeghi, editor, *FC 2013*, volume 7859 of *LNCS*, pages 371–385. Springer, Heidelberg, April 2013.

[Mea86] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, April 1986.

[MF06a] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 458–473. Springer, Heidelberg, April 2006.

[MF06b] Payman Mohassel and Matthew Franklin. Efficient polynomial operations in the shared-coefficients setting. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *PKC 2006*, volume 3958 of *LNCS*, pages 44–57. Springer, Heidelberg, April 2006.

[MG17] Sahar Mazloom and S. Dov Gordon. Differentially private access patterns in secure computation. Cryptology ePrint Archive, Report 2017/1016, 2017. http://eprint.iacr.org/2017/1016.

[NDCD$^+$13] Marcin Nagy, Emiliano De Cristofaro, Alexandra Dmitrienko, N Asokan, and Ahmad-Reza Sadeghi. Do i know you?: efficient and privacy-preserving common friend-finder protocols and applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 159–168. ACM, 2013.

[NH12] Arjun Narayan and Andreas Haeberlen. Djoin: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 149–162, 2012.

[NP99] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In S. Rao Kosaraju, editor, *12th SODA*, pages 448–457. ACM-SIAM, January 2001.

[Odl03] Andrew Odlyzko. Privacy, economics, and price discrimination on the internet. In *Proceedings of the 5th international conference on Electronic commerce*, pages 355–366. ACM, 2003.

[OG10] Femi Olumofin and Ian Goldberg. Privacy-preserving queries over relational databases. In *International Symposium on Privacy Enhancing Technologies Symposium*, pages 75–92. Springer, 2010.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 381–396. Springer, Heidelberg, February 2017.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *22nd ACM STOC*, pages 514–523. ACM Press, May 1990.

[PKV$^+$14] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos D. Keromytis, and Steve Bellovin. Blind seer: A scalable private DBMS. In *2014 IEEE Symposium on Security and Privacy*, pages 359–374. IEEE Computer Society Press, May 2014.

[PNH17] Antonis Papadimitriou, Arjun Narayan, and Andreas Haeberlen. Dstress: Efficient differentially private computations on distributed data. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 560–574. ACM, 2017.

[Pos17] Huffington Post. After court ruling: Whatsapp users can threaten abmahnkosten, 2017. http://www.huffingtonpost.de/2017/06/27/whatsapp-abmahnung-anwalt-medien-gericht-nutzer_n_17302734.html.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *European Symposium on Algorithms*, pages 121–133. Springer, 2001.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 515–530, 2015.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *Usenix Security*, volume 14, pages 797–812, 2014.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, January 2018.

[RA18] Amanda C Davi Resende and Diego F Aranha. Faster unbalanced private set intersection. *Journal of Internet Services and Applications*, 9(1):1–18, 2018.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[Res12] ABI Research. Average size of mobile games for ios increased by a whopping 42% between march and september. *London, United Kingdom*, 2012.

[Rin17] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe, 2017.

[Riv99] Ronald L. Rivest. Unconditionally secure commitment and oblivious transfer schemes using private channels and a trusted initializer, 1999. Unpublished manuscript. people.csail.mit.edu/rivest/Rivest-commitment.pdf.

[RN10] Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 735–746. ACM, 2010.

[RR16] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 297–314, Austin, TX, 2016. USENIX Association.

[RR17a] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 235–259. Springer, Heidelberg, May 2017.

[RR17b] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 17*, pages 1229–1242. ACM Press, October / November 2017.

[RS98] Martin Raab and Angelika Steger. Balls into Binsa simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[SGB18] Phillipp Schoppmann, Adri Gascn, and Borja Balle. Private nearest neighbors classification in federated databases. Cryptology ePrint Archive, Report 2018/289, 2018. https://eprint.iacr.org/2018/289.

[Smi14] Aaron Smith. 6 new facts about facebook. Pew Research Center Fact Tank, 2014. http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/.

[SV14] Nigel P Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

[TP11] Jonathan T. Trostle and Andy Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In Mike Burmester, Gene Tsudik, Spyros S. Magliveras, and Ivana Ilic, editors, *ISC 2010*, volume 6531 of *LNCS*, pages 114–128. Springer, Heidelberg, October 2011.

[WCM16] Sameer Wagh, Paul Cuff, and Prateek Mittal. Root oram: A tunable differentially private oblivious ram. *arXiv preprint arXiv:1601.03378*, 2016.